

VISUAL**DSP++**[™] **3.0**

C/C++ Compiler and Library Manual for ADSP-219x DSPs

Revision 4.0, July 2002

Part Number:
82-000390-03

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 are registered trademarks; VisualDSP++, the VisualDSP++ logo, CROSSCORE, the CROSSCORE logo, EZ-KIT Lite, Apex-ICE, Mountain-ICE, Summit-ICE, Trek-ICE, and The DSP Collaborative are trademarks of Analog Devices, Inc.

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

CONTENTS

PREFACE

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

CONTENTS

Datasheets	xxxii
Contacting DSP Publications	xxxiii
Notation Conventions	xxxiii

COMPILER

C/C++ Compiler Overview	1-2
Standard Conformance	1-4
Compiler Command-Line Interface	1-6
Running the Compiler	1-7
Specifying Compiler Options in VisualDSP++	1-10
Compiler Command-Line Switches	1-12
C/C++ Compiler Switch Summaries	1-12
C/C++ Mode Selection Switch Descriptions	1-21
-analog	1-21
-c++	1-21
-traditional	1-21
C/C++ Compiler Common Switch Descriptions	1-22
sourcefile	1-22
-@ filename	1-22
-2191	1-22
-2192-12	1-22
-219x	1-23
-A name [(<tokens>)]<="" td=""><td>1-23</td></tokens>)]>	1-23
-alttok	1-23
-build-lib	1-24

-C	1-24
-c	1-24
-Dmacro[=definition]	1-25
-debug-types	1-25
-default-linkage-{asm C C++}	1-26
-dollar	1-26
-dry	1-26
-dryrun	1-26
-E	1-26
-EE	1-27
-extra-keywords	1-27
-flags-{asm compiler lib link} switch [,switch2 [,...]]	1-27
-full-version	1-27
-g	1-28
-H	1-28
-HH	1-28
-h[elp]	1-28
-I-	1-29
-I directory [{, ;} directory...]	1-29
-include filename	1-29
-J	1-30
-L directory [{, ;} directory...]	1-30
-l library	1-30
-M	1-30

CONTENTS

-MM	1-31
-Mt filename	1-31
-MQ	1-31
-map filename	1-31
-no-alttok	1-31
-no-builtin	1-32
-no-defs	1-32
-no-dir-warnings	1-32
-no-dollar	1-32
-no-extra-keywords	1-32
-no_hardware_pc_stack	1-33
-no-inline	1-33
-no-std-def	1-33
-no-std-inc	1-33
-no-std-lib	1-34
-nothreads	1-34
-no-widen-muls	1-34
-O	1-35
-Os	1-35
-o filename	1-35
-P	1-35
-PP	1-36
-path-{ asm compiler def lib link } directory	1-36
-path-install directory	1-36

-path-output directory	1-36
-path-temp directory	1-36
-pch	1-37
-pchdir directory	1-37
-pedantic	1-37
-pedantic-errors	1-37
-pplist filename	1-38
-proc identifier	1-38
-R directory [{; }directory ...]	1-38
-R-	1-39
-reserve register[, register ...]	1-39
-S	1-39
-s	1-40
-save-temps	1-40
-show	1-40
-signed-char	1-40
-syntax-only	1-41
-sysdefs	1-41
-T filename	1-41
-threads	1-42
-time	1-42
-Umacro	1-42
-unsigned-char	1-42
-v	1-42

CONTENTS

-val-global <name-list>	1-43
-verbose	1-43
-version	1-43
-W {error remark suppress warn} number	1-43
-Wdriver-limit number	1-44
-Werror-limit number	1-44
-Wremarks	1-44
-Wterse	1-44
-w	1-44
-warn-protos	1-45
-workaround <workaround>[,<workaround>]*	1-45
-write-files	1-45
-write-opts	1-45
-xref <filename>	1-45
C++ Mode Compiler Switch Descriptions	1-46
-bool	1-46
-explicit	1-46
-force	1-47
-instant{all local used}	1-47
-namespace	1-48
-newforinit	1-48
-newvec	1-48
-no-bool	1-48
-no-demangle	1-48

-no-explicit	1-49
-no-namespace	1-49
-no-newvec	1-49
-no-restrict	1-49
-notstrict	1-49
-no-wchar	1-49
-restrict	1-50
-strict	1-50
-strictwarn	1-50
-suppress	1-50
-tpautooff	1-51
-trdforinit	1-51
-typename	1-51
-wchar	1-51
Data Type Sizes	1-52
C/C++ Compiler Language Extensions	1-54
Inline Function Support Keyword (inline)	1-57
Inline Assembly Language Support Keyword (asm)	1-58
Assembly Construct Template	1-59
ASM() Construct Syntax:	1-59
ASM() Construct Syntax Rules	1-61
ASM() Construct Template Example	1-62
Assembly Construct Operand Description	1-62
Assembly Constructs with Multiple Instructions	1-65

CONTENTS

Assembly Construct Reordering and Optimization	1-66
Assembly Constructs with Input and Output Operands	1-67
Assembly Constructs and Macros	1-68
Dual Memory Support Keywords (pm dm)	1-69
Memory Keywords and Assignments/Type Conversions	1-71
Memory Keywords and Function Declarations/Pointers	1-72
Memory Keywords and Function Arguments	1-73
Memory Keywords and Macros	1-74
Placement Support Keyword (section)	1-74
Boolean Type Support Keywords (bool, true, false)	1-75
Pointer Class Support Keyword (restrict)	1-75
Variable Length Array Support	1-76
Non-Constant Aggregate Initializer Support	1-78
Indexed Initializer Support	1-78
Aggregate Constructor Expression Support	1-81
Fractional Type Support	1-82
Format of Fractional Literals	1-82
Conversions Involving Fractional Values	1-83
Fractional Arithmetic Operations	1-83
Mixed Mode Operations	1-84
Saturated Arithmetic	1-84
Preprocessor Generated Warnings	1-85
C++ Style Comments	1-85
Compiler Built-in Functions	1-85

Access to System Registers	1-86
I/O Space Read or Write	1-88
Interrupt Control	1-89
Mode Control	1-90
External Memory Read or Write	1-90
ETSI Support	1-91
ETSI Support Overview	1-92
Calling ETSI Library Functions	1-94
Using the ETSI Built-In Functions	1-95
Linking ETSI Library Functions	1-95
Working with ETSI Library Source Code	1-96
ETSI Support for Data Types	1-96
ETSI Header File	1-97
Pragmas	1-99
Data Alignment Pragma - ALIGN NUM	1-100
Interrupt Handler Pragma	1-100
Altregisters Pragma	1-101
Optimization Level Pragmas	1-101
Linkage Pragmas	1-101
linkage_name Identifier	1-102
weak_entry Pragma	1-102
Preprocessor Features	1-103
Predefined Preprocessor Macros	1-103
Header Files	1-105

CONTENTS

Writing Macros	1-105
Preprocessing of .IDL Files	1-107
C/C++ Run-Time Model and Environment	1-109
Using the Run-Time Header	1-109
Interrupt Table and Interface	1-110
Stack Frame	1-111
Stack Frame Description	1-113
General System-Wide Specifications	1-114
At a procedure call, the following must be true:	1-115
At an interrupt, the following must be true:	1-115
Return Values	1-115
Procedure Call and Return	1-115
On Entry:	1-116
To Return from a Procedure:	1-116
Miscellaneous Information	1-117
Register Classification	1-117
Callee Preserved Registers (“Preserved”)	1-117
Dedicated Registers	1-117
Caller Save Registers (“Scratch”)	1-117
Circular Buffer Length Registers	1-118
Mode Status (MSTAT) Register	1-118
Complete List of Registers	1-119
C/C++ and Assembly Language Interface	1-123
Calling Assembly Subroutines from C/C++ Programs	1-123

Calling C/C++ Functions from Assembly Programs	1-126
Using Mixed C/C++ and Assembly Naming Conventions	1-127
C++ Programming Examples	1-129
Using Fract Type Support	1-129
Using Complex Number Support	1-130

C/C++ RUN-TIME LIBRARY

C and C++ Run-Time Libraries Guide	2-3
Calling Library Functions	2-3
Using the Compiler's Built-In C Library Functions	2-4
Linking Library Functions	2-5
Working With Library Source Code	2-7
Working With Library Header Files	2-8
C Run-Time Library Header File Descriptions	2-9
assert	2-9
ctype	2-9
def2191 – Memory Map Definitions	2-10
def2192-12 – Memory Map Definitions	2-10
def219x – Memory Map Definitions	2-10
errno	2-10
float	2-10
limits	2-11
locale	2-11
math	2-11
setjmp	2-12

CONTENTS

signal	2-12
stdarg	2-12
stddef	2-12
stdio	2-12
stdlib	2-13
string	2-13
sysreg	2-13
Abridged C++ Library Support	2-14
Embedded C++ Library Header Files	2-14
complex	2-14
exception	2-14
fract	2-15
fstream	2-15
iomanip	2-15
ios	2-15
iosfwd	2-15
iostream	2-15
istream	2-16
new	2-16
ostream	2-16
sstream	2-16
stdexcept	2-16
streambuf	2-16
string	2-17

strstream	2-17
C++ Header Files for Library Facilities	2-17
Embedded Standard Template Library Header Files	2-18
algorithm	2-18
deque	2-18
functional	2-19
hash_map	2-19
hash_set	2-19
iterator	2-19
list	2-19
map	2-19
memory	2-19
numeric	2-19
queue	2-19
set	2-20
stack	2-20
utility	2-20
vector	2-20
fstream.h	2-20
iomanip.h	2-20
iostream.h	2-20
new.h	2-20
Documented Library Functions	2-21
C Run-Time Library Reference	2-24

CONTENTS

Notation Conventions	2-24
Reference Format	2-24
abort	2-25
abs	2-26
acos	2-27
asin	2-28
atan	2-29
atan2	2-30
atexit	2-31
atof	2-32
atoi	2-33
atol	2-34
bsearch	2-35
calloc	2-37
ceil	2-38
clear_interrupt	2-39
cos	2-41
cosh	2-42
disable_interrupts	2-43
div	2-44
enable_interrupts	2-45
exit	2-46
exp	2-47
external_memory_read	2-48

<code>external_memory_write</code>	2-50
<code>fabs</code>	2-52
<code>floor</code>	2-53
<code>fmod</code>	2-54
<code>free</code>	2-55
<code>frexp</code>	2-56
<code>interrupt</code>	2-57
<code>io_space_read</code>	2-61
<code>io_space_write</code>	2-62
<code>isalnum</code>	2-63
<code>isalpha</code>	2-64
<code>iscntrl</code>	2-65
<code>isdigit</code>	2-66
<code>isgraph</code>	2-67
<code>isinf</code>	2-68
<code>islower</code>	2-70
<code>isnan</code>	2-71
<code>isprint</code>	2-73
<code>ispunct</code>	2-74
<code>isspace</code>	2-75
<code>isupper</code>	2-76
<code>isxdigit</code>	2-77
<code>labs</code>	2-78
<code>ldexp</code>	2-79

CONTENTS

ldiv	2-80
log	2-81
log10	2-82
longjmp	2-83
malloc	2-85
memchr	2-86
memcmp	2-87
memcpy	2-88
memcpy_from_shared	2-89
memcpy_to_shared	2-90
memmove	2-91
memset	2-92
mode_change	2-93
modf	2-95
pow	2-96
qsort	2-97
raise	2-99
rand	2-101
realloc	2-102
setjmp	2-103
signal	2-105
sin	2-108
sinh	2-109
sqrt	2-110

<code>srand</code>	2-111
<code>strcat</code>	2-112
<code>strchr</code>	2-113
<code>strcmp</code>	2-114
<code>strcoll</code>	2-115
<code>strcpy</code>	2-116
<code>strcspn</code>	2-117
<code>strerror</code>	2-118
<code>strlen</code>	2-119
<code>strncat</code>	2-120
<code>strncmp</code>	2-121
<code>strncpy</code>	2-122
<code>strpbrk</code>	2-123
<code>strrchr</code>	2-124
<code>strspn</code>	2-125
<code>strstr</code>	2-126
<code>strtod</code>	2-127
<code>strtodf</code>	2-129
<code>strtok</code>	2-131
<code>strtol</code>	2-133
<code>strtoul</code>	2-135
<code>strxfrm</code>	2-137
<code>sysreg_read</code>	2-139
<code>sysreg_write</code>	2-141

tan	2-143
tanh	2-144
tolower	2-145
toupper	2-146
va_arg	2-147
va_end	2-149
va_start	2-150

DSP RUN-TIME LIBRARY

DSP Run-Time Library Guide	3-2
Calling DSP Library Functions	3-2
Linking DSP Library Functions	3-3
Working with Library Source Code	3-3
DSP Header Files	3-4
complex.h — Basic Complex Arithmetic Functions	3-4
filter.h — DSP Filters and Transformations	3-6
math.h — Math Functions	3-10
matrix.h — Matrix Functions	3-12
stats.h — Statistical Functions	3-15
vector.h — Vector Functions	3-16
window.h — Window Generators	3-20
DSP Run-Time Library Reference	3-21
Notation Conventions	3-21
a_compress	3-22
a_expand	3-23

arg	3-24
autocoh	3-25
autocorr	3-26
cabs	3-27
cadd	3-28
cdiv	3-29
cexp	3-30
cfft	3-31
cfft4	3-33
cfft2d	3-35
cfir	3-37
clip	3-39
cmlt	3-40
conj	3-41
convolve	3-42
conv2d	3-44
conv2d3x3	3-45
copysign	3-46
cot	3-47
countones	3-48
crosscoh	3-49
crosscorr	3-50
csub	3-51
fir	3-52

fir_decima	3-54
fir_interp	3-56
gen_bartlett	3-58
gen_blackman	3-60
gen_gaussian	3-61
gen_hamming	3-62
gen_hanning	3-63
gen_harris	3-64
gen_kaiser	3-65
gen_rectangular	3-66
gen_triangle	3-67
gen_vonhann	3-69
histogram	3-70
ifft	3-71
iffttrad4	3-73
ifft2d	3-75
iir	3-77
max	3-79
mean	3-80
min	3-81
mu_compress	3-82
mu_expand	3-83
norm	3-84
polar	3-85

rfft	3-86
rfftrad4	3-88
rfft2d	3-90
rms	3-92
rsqrt	3-93
twidfftrad2	3-94
twidfftrad4	3-96
twidfft2d	3-98
var	3-100
zero_cross	3-101

COMPILER LEGACY SUPPORT

Tools Differences	A-1
C/C++ Compiler and Run-time Library	A-3
Segment Placement Support Keyword Changed to Section	A-3
G21 Compatibility Call	A-3
Support for G21-Based Options And Extensions	A-4
ANSI C Extensions	A-4
Compiler Switch Modifications	A-5
New and Obsolete Warnings	A-8
Run-Time Model	A-9
C/C++ Run-Time Library	A-9

INDEX

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 ADSP-219x DSPs* contains information about the C/C++ compiler and run-time library program for ADSP-219x 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 ADSP-219x DSPs architecture and instruction set and the C/C++ instruction set.

Programmers who are unfamiliar with ADSP-219x 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 ADSP-219x 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 ADSP-219x DSPs documents support for all ADSP-219x 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 other enhancements.

Technical or Customer Support

You can reach DSP Tools Support in the following ways:

- Visit the DSP Development Tools website at
www.analog.com/technology/dsp/developmentTools/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 “ADSP-219x” refers to a family of Analog Devices 16-bit, fixed-point processors. VisualDSP++ currently supports the following ADSP-219x processors:

ADSP-2191 DSP, ADSP-2192-12 DSP, ADSP-2195 DSP,
ADSP-2196 DSP, ADSP-21990 DSP, ADSP-21991 DSP, and
ADSP-21992 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` 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 ADSP-219x DSPs

VisualDSP++ 3.0 User's Guide for ADSP-219x DSPs

VisualDSP++ 3.0 C/C++ Compiler and Library Manual for ADSP-219x DSPs

VisualDSP++ 3.0 C/C++ Assembler and Preprocessor Manual for ADSP-218x and ADSP-219x DSPs

VisualDSP++ 3.0 Linker and Utilities Manual for ADSP-218x and ADSP-219x 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.
.HTM or .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 (toolbar 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.

From the Web

To download the tools manuals, point your browser at 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 phone 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> .
[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> .

Notation Conventions

Example	Description
<code>.SECTION</code>	Commands, directives, keywords, and feature names are in text with letter gothic font.
<code>filename</code>	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 (`cc219x`) is part of Analog Devices development software for ADSP-219x DSPs.

This chapter contains:

- [“C/C++ Compiler Overview” on page 1-2](#)
Provides an overview of C/C++ compiler for ADSP-219x DSPs.
- [“Compiler Command-Line Interface” on page 1-6](#)
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 1-54](#)
Describes the `cc219x` compiler’s extensions to the ISO/ANSI standard for the C and C++ languages.
- [“Preprocessor Features” on page 1-103](#)
Contains information on the preprocessor and ways to modify source compilation.
- [“C/C++ Run-Time Model and Environment” on page 1-109](#)
Contains reference information about implementation of C/C++ programs, data, and function calls in ADSP-219x DSPs.
- [“C/C++ and Assembly Language Interface” on page 1-123](#)
Describes how to call an assembly language subroutine from 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 (`cc219x`) 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 (`cc219x`) compiles ISO/ANSI standard C and C++ code for the ADSP-219x DSPs. Additionally, Analog Devices includes within the compiler a number of C language extensions designed to assist in DSP development. The `cc219x` compiler runs from the VisualDSP++ environment or from an operating system command line.

The C/C++ compiler (`cc219x`) processes your C and C++ language source files and produces ADSP-219x DSP's assembler source files. The assembler source files are assembled by the ADSP-219x assembler (`easm219x`). The assembler creates Executable and Linkable Format (ELF) object files that can either be linked (using the linker) to create an ADSP-219x executable file or included in an archive library (using `elfar`). The way in which the compiler controls the assemble, link, and archive phases of the process depends on the source input files and the compiler options used.

Source files contain the C/C++ program to be processed by the compiler. The `cc219x` compiler supports the ANSI/ISO standard definitions of the C and C++ languages. For information on the C language standard, see

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

The cc219x compiler supports the proposed Embedded C++ Standard. The Embedded C++ Standard 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, cc219x supports the proposed standard definition, templates, and all other 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 the compiler performance.

The cc219x compiler supports a set of C/C++ language extensions. These extensions support hardware features of the ADSP-219x DSPs. For information on these extensions, see [“C/C++ Compiler Language Extensions” on page 1-54](#).

Compiler options are set in the VisualDSP++ Integrated Development and Debug Environment (IDDE) from the **Compile** page of the **Project Options** dialog box (see [“Specifying Compiler Options in VisualDSP++” on page 1-10](#)). The 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 ADSP-21xx DSPs* and online Help.

Standard Conformance

Analog C compilers conform to the ISO/IEC 998:1990 C standard and the ISO/IEC 14882:1998 C++ standard (in C++ mode) with a small number of currently unsupported features or areas of divergence.

Unsupported features are:

- Runtime-type information (RTTI) for C++ is not supported.
- Exceptions for C++ is not supported.
- ANSI features that require operating-system support are generally not supported. This includes `time.h` functionality in C.
- The `cc219x` compiler does not provide comprehensive support of NaN's, overflow and underflow conditions in their compiler support floating-point routines.

Areas of divergence from Standard:

- The `double` type is defined in terms of a single precision 32-bit floats, not double precision 64-bit floats.
- The `cc219x` compiler makes use of the DSP's double word (long) MAC instruction results to avoid having to explicitly promote integer operand multiplication to long. If the integer multiplication result overflows the integer type, then the result is not truncated as would be the case in strict ANSI terms. This behaviour is disabled using the compiler's "[-no-widen-muls](#)" on page 1-34.
- Normal ANSI C external linkage does not specifically require standard include files to be used, although it is recommended. In most cases, Analog C compilers do require standard include files to be used. This is because build configurations and optimization levels are used to select the correct and optimal implementation of the functions. For example, the include files may redefine standard C functions to use optimal compiler built-in implementations.

The compilers also support a number of language extensions that are essentially aids to DSP programmers and would not be defined in strict ANSI conforming implementations. These extensions are usually enabled by default and in some cases can be disabled using a command-line switch, if required.

These extensions include:

- Inline (function) which directs the compiler to integrate the function code into the code of the callers. Disabled using the `-no-inline` compiler switch.
- Dual memory support keywords (`pm/dm`). Disabled using the `-no-extra-keywords` compiler switch (see [on page 1-32](#)).
- Placement support keyword (`section`). Disabled using the `-no-extra-keywords` compiler switch (see [on page 1-32](#)).
- Boolean type support keywords in C (`bool`, `true`, `false`). Disabled using the `-no-extra-keywords` compiler switch (see [on page 1-32](#)).
- Variable length array support
- Non-constant aggregate initializer support
- Indexed initializer support
- Preprocessor generated warnings
- Support for C++-style comments in C programs

For more information on these extensions, see the “[C/C++ Compiler Language Extensions](#)” on [page 1-54](#).

Compiler Command-Line Interface

This section describes how the `cc219x` 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 1-7](#)
- [“Specifying Compiler Options in VisualDSP++” on page 1-10](#)
- [“Compiler Command-Line Switches” on page 1-12](#)
- [“Data Type Sizes” on page 1-52](#)

By default, the compiler runs with Analog Extensions for C code enabled. This means that the compiler processes source files written in ANSI/ISO standard C language supplemented with 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 `.DXE` file (see file names in [Table 1-2 on page 1-10](#)). [Table 1-3 on page 1-12](#) lists the switches that select the language dialect.

Although many switches are generic between C and C++, some of them are valid in C++ mode only. A summary of the generic C/C++ compiler switches appears in [Table 1-4 on page 1-13](#). A summary of the C++-specific compiler switches appears in [Table 1-5 on page 1-19](#). 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 1-10](#) for more information.

Running the Compiler

Use the following general syntax for the `cc219x` command line:

```
cc219x [-switch [-switch ...]] sourcefile [sourcefile ...]
```

where:

- `-switch` is the name of the switch to be processed. The compiler has many switches. These switches select the operations and modes for the compiler and other tools. Command-line switches are case sensitive. For example, `-O` is not the same as `-o`.
- `sourcefile` 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, using either forward or back slashes as the directory delimiter. It also supports UNC path names (starting with two slashes and a network name).

The `cc219x` compiler uses the file extension to determine what the file contains and what operations to perform upon it. [Table 1-2 on page 1-10](#) lists the allowed extensions.

For example, the following command line

```
cc219x -2191 -O -Wremarks -o program.dxe source.c
```

runs `cc219x` with the following switches:

<code>-2191</code>	Specifies the processor
<code>-O</code>	Specifies optimization for the compiler
<code>-Wremarks</code>	Selects extra diagnostic remarks in addition to warning and error messages
<code>-o program.dxe</code>	Selects a name for the compiled, linked output

Compiler Command-Line Interface

`source.c` Specifies the C language source file to be compiled

The following example command line

```
cc219x -2191 -c++ source.cpp
```

runs `cc219x` with:

`-c++` Specifies that all of the source files are written in C++

`source.cpp` Specifies the C++ language source file for your program

The normal function of `cc219x` 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 filenames, and by various switches.

In normal operation the compiler uses the following extension files to perform a specified action:

Table 1-1. File Extensions

Extension	Action
<code>.C</code> <code>.c</code> <code>.cpp</code> <code>.cxx</code>	Source file is compiled, assembled, and linked
<code>.asm</code> , <code>.dsp</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 first processed to produce an object file; then all object files are presented to the linker.

You can stop this sequence at various points by using appropriate compiler switches, or by selecting options with the VisualDSP++ environment. These switches are `-E`, `-P`, `-M`, `-H`, `-S`, and `-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, `cc219x` names the output for you. [Table 1-2 on page 1-10](#) lists the type of files, names, and extensions `cc219x` 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, any search directories that you select, and any path information that you include in the file name. [Table 1-2](#) indicates the searches that the preprocessor, compiler, assembler, and linker support. The compiler supports relative and absolute directory names to define file search paths. For information on additional search directories, see the `-I` directory switch ([on page 1-29](#)) and `-L` directory switch ([on page 1-30](#)).

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 `cc219x` compiler uses the file extension conventions listed in [Table 1-2](#) to determine the input file type.

- Verify that the compiler is using the correct file. If you do not provide the complete file path as part of the parameter or add additional search directories, `cc219x` looks for input in the current directory.



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

Compiler Command-Line Interface

Table 1-2. Input and Output Files

Input File Extension	File Extension Description
.c	C/C++ source file.
.C .cpp .cxx	C++ source file.
.h	Header file (referenced by a <code>#include</code> statement).
.pch	C++ pre-compiled header file.
.i	Preprocessed C/C++ source, created when preprocess only (<code>-E</code> compiler switch) is specified.
.s, .dsp, .asm	Assembler source file.
.idl	Interface definition language files for VCSE.
.ii	Template instantiation files -- used internally by the compiler when instantiating C++ templates
.is	Preprocessed assembly 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.
.map	DSP system memory map file output.
.sym	DSP system symbol map file output.

Specifying Compiler Options in VisualDSP++

When using the VisualDSP++ Integrated Development and Debug Environment (IDDE), use the **Compile** page from the **Project Options** dialog box to set compiler functional options as shown on [Figure 1-1](#).

Callouts refer to the corresponding compiler command-line switches described in [“Compiler Command-Line Switches” on page 1-12](#). 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. Use the VisualDSP++ online Help to get more information on compiler options you can specify from the VisualDSP++ environment.

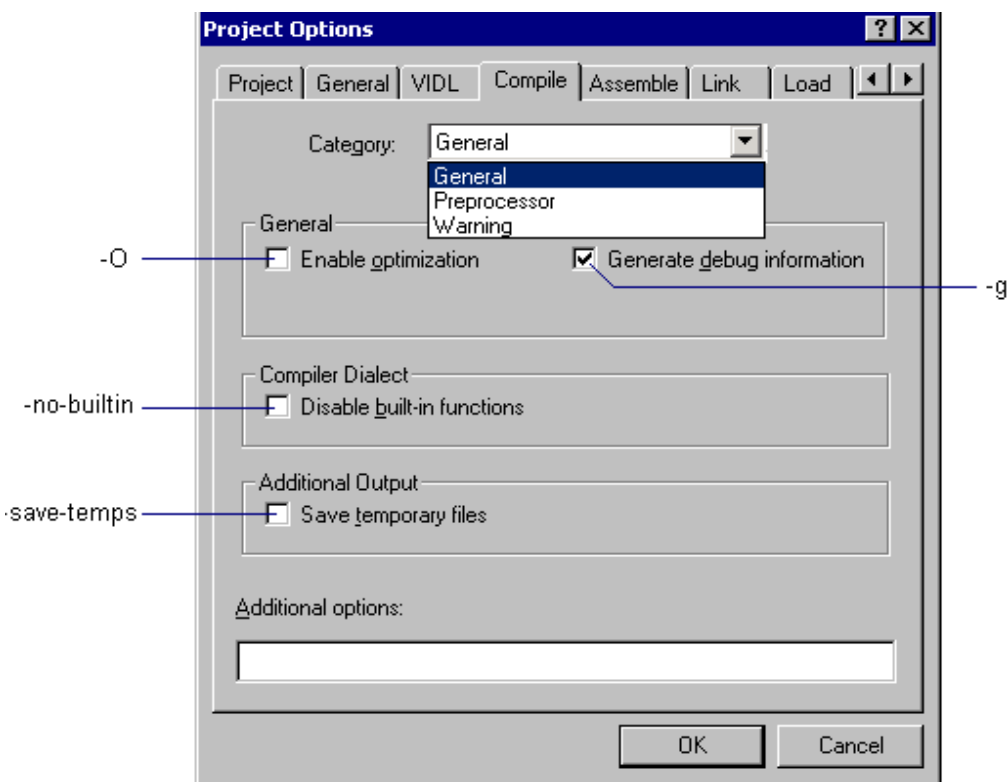


Figure 1-1. Project Options – Compile Property Page

Compiler Command-Line Switches

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

C/C++ Compiler Switch Summaries

This section contains a set of tables that summarize generic and specific switches (options), as follows:

- “C or C++ Mode Selection Switches” [Table 1-3 on page 1-12](#)
- “C/C++ Compiler Common Switches” [Table 1-4 on page 1-13](#)
- “C++ Mode Compiler Switches” [Table 1-5 on page 1-19](#).

A brief description of each switch follows the tables, beginning on [page 1-21](#).

Table 1-3. C or C++ Mode Selection Switches

Switch Name	Description
<code>-analog</code> (on page 1-21)	Supports ANSI/ISO standard C with Analog Devices extensions. Default mode.
<code>-++c</code> (on page 1-21)	Supports ANSI/ISO standard C++ with Analog Devices extensions.
<code>-traditional</code> (on page 1-21)	Supports pre-ANSI K&R C.

Table 1-4. C/C++ Compiler Common Switches

Switch Name	Description
<code>sourcefile</code> (on page 1-22)	Specifies file to be compiled.
<code>-@ filename</code> (on page 1-22)	Reads command-line input from the file.
<code>-21{9x 91 92-12}</code> (on page 1-22)	Instructs the compiler to produce code suitable for the specified ADSP-219x DSP.
<code>-A name[tokens]</code> (on page 1-23)	Asserts the specified name as a predicate.
<code>-alttok</code> (on page 1-23)	Allows alternative keywords and sequences in sources.
<code>-build-lib</code> (on page 1-24)	Directs the librarian to build a library file.
<code>-C</code> (on page 1-24)	Retains preprocessor comments in the output file; active only with the <code>-E</code> or <code>-P</code> switch).
<code>-c</code> (on page 1-24)	Compiles and/or assembles only; does not link.
<code>-Dmacro[=definition]</code> (on page 1-25)	Defines a macro.
<code>-debug-types</code> (on page 1-25)	Supports building a *.h file directly and writing a complete set of debugging information for the header file.
<code>-default-linkage-{asm C C++}</code> (on page 1-26)	Sets the default linkage type (C, C++, asm).
<code>-dollar</code> (on page 1-26)	Accepts dollar signs in symbols.
<code>-dry</code> (on page 1-26)	Displays, but does not perform, main driver actions (verbose dry-run).
<code>-dryrun</code> (on page 1-26)	Displays, but does not perform, top-level driver actions (terse dry-run).
<code>-E</code> (on page 1-26)	Preprocesses, but does not compile, the source file.

Compiler Command-Line Interface

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

Switch Name	Description
-EE (on page 1-27)	Preprocesses and compiles the source file.
-extra-keywords (on page 1-27)	Recognizes Analog Devices extensions to ISO/ANSI standards for C and C++. Default mode.
-flags-{tools} <arg1> [,arg2...] (on page 1-27)	Passes command-line switches through the compiler to other build tools.
-full-version (on page 1-27)	Displays version information for build tools.
-g (on page 1-28)	Generates DWARF-2 debug information.
-H (on page 1-28))	Outputs a list of header files, but does not compile the source file.
-HH (on page 1-28)	Outputs a list of included header files and compiles.
-h[elp] (on page 1-28)	Outputs a list of command-line switches with brief syntax descriptions.
-I- (on page 1-29)	Establishes the point in the <code>include</code> directory list at which the search for header files enclosed in angle brackets should begin.
-I <i>directory</i> (on page 1-29)	Appends <i>directory</i> to the standard search path.
-include <i>filename</i> (on page 1-29)	Includes named file prior to preprocessing each source file.
-J (on page 1-30)	Performs "old-style" preprocessing.
-L <i>directory</i> (on page 1-30)	Appends the specified directory to the standard library search path when linking.
-l <i>library</i> (on page 1-30)	Searches the specified library for functions when linking.

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

Switch Name	Description
-M (on page 1-30)	Generates make rules only; does not compile.
-MM (on page 1-31)	Generates make rules and compiles.
-Mt <i>filename</i> (on page 1-31)	Makes dependencies for the specified source file.
-MQ (on page 1-31)	Generates make rules only; does not compile. No notification when input files are missing.
-map <i>filename</i> (on page 1-31)	Directs the linker to generate a memory map of all symbols.
-no-builtin (on page 1-32)	Disables recognition of <code>__builtin</code> functions.
-no-defs (on page 1-32)	Does not define any default preprocessor macros, include directories, library directories, libraries, run-time headers, or keyword extensions.
-no-dir-warnings (on page 1-32)	Disables warnings about include and library switch directories that do not exist.
-no-dollar (on page 1-32)	Does not accept dollar signs in symbols.
-no-extra-keywords (on page 1-32)	Does not define language extension keywords that could be valid C or C++ identifiers.
-no_hardware_pc_stack (on page 1-33)	Uses software stack instead of default hardware stack for return PC.
-no-inline (on page 1-33)	Ignores the <code>inline</code> keyword.
-no-std-def (on page 1-33)	Disables normal macro definitions; also disables Analog Devices keyword extensions that do not have leading underscores (<code>__</code>).
-no-std-inc (on page 1-33)	Searches for preprocessor header files only in the current directory and in directories specified with the <code>-I</code> switch.

Compiler Command-Line Interface

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

Switch Name	Description
<code>-no-std-lib</code> (on page 1-34)	Searches for only those linker libraries specified with the <code>-l</code> switch when linking.
<code>-nothreads</code> (on page 1-34)	Specifies that compiled code does not need to be thread-safe.
<code>-no-widen-muls</code> (on page 1-34)	Disable widening multiplications optimization.
<code>-O</code> (on page 1-35)	Enables optimizations.
<code>-Os</code> (on page 1-35)	Optimizes for code size.
<code>-o filename</code> (on page 1-35)	Specifies the output file name.
<code>-P</code> (on page 1-35)	Preprocesses, but does not compile, the source file. Omits line numbers in the preprocessor output.
<code>-PP</code> (on page 1-36)	Similar to <code>-P</code> , but does not halt compilation after preprocessing.
<code>-path-install directory</code> (on page 1-36)	Uses the specified directory as the location for all compilation tool.
<code>-path-output directory</code> (on page 1-36)	Specifies the location of non-temporary files.
<code>-path-temp directory</code> (on page 1-36)	Specifies the location of temporary files.
<code>-path-tool directory</code> (on page 1-36)	Uses the specified directory as the location of the specified compilation tool (assembler, compiler, driver definitions file, library builder, or linker).
<code>-pch</code> (on page 1-37)	Generates and uses precompiled header files (*.pch)
<code>-pchdir directory</code> (on page 1-37)	Specifies the location of PCHRepository.
<code>-pedantic</code> (on page 1-37)	Issues compiler warnings for any constructs that are not ISO/ANSI standard C/C++-compliant.

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

Switch Name	Description
<code>-pedantic-errors</code> (on page 1-37)	Issues compiler errors for any constructs that are not ISO/ANSI standard C/C++-compliant.
<code>-pplist filename</code> (on page 1-38)	Outputs a raw preprocessed listing to the specified file.
<code>-proc identifier</code> (on page 1-38)	Specifies that the compiler should produce code suitable for the specified DSP.
<code>-R directory</code> (on page 1-38)	Appends directory to the standard search path for source files.
<code>-R-</code> (on page 1-39)	Removes all directories from the standard search path for source files.
<code>-reserve <reg1>[,reg2...]</code> (on page 1-39)	Reserves the I2, I3, and M0 registers from compiler use. Note: Reserving registers can have a detrimental effect on the compiler's optimization capabilities.
<code>-S</code> (on page 1-39)	Stops compilation before running the assembler.
<code>-s</code> (on page 1-40)	Removes debugging information from the output executable file when linking.
<code>-save-temps</code> (on page 1-40)	Saves intermediate compiler temporary files.
<code>-show</code> (on page 1-40)	Displays the driver command-line information.
<code>-signed-char</code> (on page 1-40)	Makes the default type for <code>char</code> signed.
<code>-syntax-only</code> (on page 1-41)	Checks the source code for compiler syntax errors, but does not write any output.
<code>-sysdefs</code> (on page 1-41)	Defines the system definition macros.
<code>-T filename</code> (on page 1-41)	Uses the specified the Linker Description File as control input for linking.
<code>-threads</code> (on page 1-42)	Specifies that the build and link should be thread-safe.

Compiler Command-Line Interface

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

Switch Name	Description
<code>-time</code> (on page 1-42)	Displays the elapsed time as part of the output information on each part of the compilation process.
<code>-traditional</code> (on page 1-21)	Applies traditional C compiler rules (consistent with pre-ANSI K&R C compilers).
<code>-Umacro</code> (on page 1-42)	Undefines macro (s).
<code>-unsigned-char</code> (on page 1-42)	Makes the default type for <code>char</code> unsigned.
<code>-v</code> (on page 1-42)	Displays both the version and command-line information for all compilation tools as they process each file (version & verbose).
<code>-val-global <name-list></code> (on page 1-43)	Adds global names.
<code>-verbose</code> (on page 1-43)	Displays command-line information for all compilation tools as they process each file.
<code>-version</code> (on page 1-43)	Displays version information for all compilation tools as they process each file.
<code>-W{error remark suppress warn} number</code> (on page 1-43)	Overrides the default severity of the specified diagnostic-messages (errors, remarks, or warnings)..
<code>-Wdriver-limit number</code> (on page 1-43)	Aborts the driver after reaching the specified number of errors.
<code>-Werror-limit number</code> (on page 1-44)	Stops compiling after reaching the specified number of errors.
<code>-Wremarks</code> (on page 1-44)	Indicates that the compiler may issue remarks, which are diagnostic messages even milder than warnings.
<code>-Wterse</code> (on page 1-44)	Issues only the briefest form of compiler warnings, errors, and remarks.
<code>-w</code> (on page 1-44)	Does not display compiler warning messages.

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

Switch Name	Description
<code>-warn-protos</code> (on page 1-45)	Produces a warning when a function is called without a prototype.
<code>-workaround <workaround></code> <code>[,<workaround>]*</code> (on page 1-45)	Enables code generator workaround for specific hardware defects.
<code>-write-files</code> (on page 1-45)	Enables compiler I/O redirection.
<code>-write-opts</code> (on page 1-45)	Passes the user options (but not input filenames) via a temporary file.
<code>-xref filename</code> (on page 1-45)	Outputs cross-reference information to the specified file.

Table 1-5. C++ Mode Compiler Switches

Switch Name	Description
<code>-bool</code> (on page 1-46)	Enables the <code>bool</code> keyword.
<code>-explicit</code> (on page 1-46)	Supports the <code>explicit</code> specifier on constructor declarations.
<code>-force</code> (on page 1-47)	Forces virtual table definitions.
<code>-instant{all local used}</code> (on page 1-47)	Instantiates all, local or used members of a class.
<code>-namespace</code> (on page 1-48)	Supports namespaces.
<code>-newforinit</code> (on page 1-48)	Limits the scope of any symbol declared within a “for” statement.
<code>-newvec</code> (on page 1-48)	Allows the overloading of <code>new</code> and <code>delete</code> .
<code>-no-attok</code> (on page 1-31)	Does not allow alternative keywords and sequences in sources.

Compiler Command-Line Interface

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

Switch Name	Description
<code>-no-bool</code> (on page 1-48)	Disables the <code>bool</code> keyword.
<code>-no-demangle</code> (on page 1-48)	Prevents filtering of any linker errors through the demangler.
<code>-no-explicit</code> (on page 1-49)	Does not support the <code>explicit</code> specifier on constructor declarations.
<code>-no-namespace</code> (on page 1-49)	Does not support namespaces.
<code>-no-newvec</code> (on page 1-49)	Does not allow the overloading of <code>new</code> and <code>delete</code> .
<code>-no-restrict</code> (on page 1-49)	Disables the <code>restrict</code> keyword.
<code>-notstrict</code> (on page 1-49)	Omits warning and/or error messages for non-ANSI constructs.
<code>-no-wchar</code> (on page 1-49)	Disables <code>new wchar_t</code> .
<code>-restrict</code> (on page 1-50)	Enables the <code>restrict</code> keyword.
<code>-strict</code> (on page 1-50)	Generates error messages for non-ANSI constructs.
<code>-strictwarn</code> (on page 1-50)	Generates warning messages for non-ANSI constructs.
<code>-suppress</code> (on page 1-50)	Suppresses virtual table definitions.
<code>-tpautooff</code> (on page 1-51)	Disables automatic instantiation of templates.
<code>-trdforinit</code> (on page 1-51)	Limits the scope of any symbol declared within a “for” statement.

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

Switch Name	Description
<code>-typename</code> (on page 1-51)	Recognizes the <code>typename</code> keyword.
<code>-wchar</code> (on page 1-51)	Enables new <code>wchar_t</code> .

C/C++ Mode Selection Switch Descriptions

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

-analog

The `-analog` (Analog C compilation) switch directs the compiler to support Analog Devices extensions to ISO/ANSI standard C. This is the default mode. For information on these extensions, see “[C/C++ Compiler Language Extensions](#)” on page 1-54 .

-c++

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

-traditional

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

- All `extern` declarations (including implicit declarations of functions) take effect globally.
- The keywords `inline`, `signed`, `const` and, `volatile` are not recognized.
- 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) switch 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 cc219x compiler uses the file extension to determine the operations to perform. [Table 1-2 on page 1-10](#) lists the permitted extensions and matching compiler operations.

-@ filename

The @ filename (command file) switch directs the compiler to read command-line input from *filename*. The specified *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.

-2191

The -2191 (compile for ADSP-2191 DSP) switch directs the build tools chain to generate code suitable for the ADSP-2191 DSP. When compiling with this switch, the __ADSP21XX__, __ADSP219X__, and __ADSP2191__ preprocessor macros are defined as 1.

-2192-12

The -2192-12 (compile for ADSP-2192-12 DSP) switch directs the build tools chain to generate code suitable for the ADSP-2192-12 DSP. When compiling with this switch, the __ADSP21XX__, __ADSP219X__, and __ADSP2192_12__ preprocessor macros are defined as 1.

-219x

The `-219x` (compile for ADSP-219x DSPs) switch directs the build tools chain to generate code suitable for the ADSP-219x DSPs; the `-219x` switch results in code suitable for any chip of the ADSP-219x DSPs. When compiling with this switch, the C compiler defines the corresponding `__ADSP219X__` macro as well as the `__ADSP21XX__` macro.

-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	<code>embedded</code>
Machine	<code>adsp219x</code>
CPU	<code>adsp219x</code>
Compiler	<code>cc219x</code>

-alttok

The `-alttok` (alternative tokens) switch directs the compiler to allow alternative operator keywords and digraph sequences in source files. This is the default mode. The “`-no-alttok`” switch ([on page 1-31](#)) can be used to disallow such support.

The ANSI C trigraphs sequences are always expanded (even with the `-no-alttok` option), and only digraph sequences are expanded 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>

Compiler Command-Line Interface

Keyword	Equivalent
bitor	
compl	~
not	!
not_eq	!=
or	
or_eq	=
xor	^
xor_eq	^=



To use them in C, you should use `#include <iso646.h>.`

-build-lib

The `-build-lib` (build library) switch directs the compiler to use `elfar` (librarian) to produce a library file (`.dlb`) as the output 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.

-c

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

-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, 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.

-debug-types

The `-debug-types` switch 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` switch because it is implied.

For example,

```
cc219x -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`.

Compiler Command-Line Interface

-default-linkage-{asm|C|C++}

The `-default-linkage-asm` (assembler linkage)/`-default-linkage-C` (C linkage)/`-default-linkage-C++` (C++ linkage) switch directs the compiler to set the default linkage type. C linkage is the default type in C mode, and C++ linkage is the default type in C++ mode.

 This switch can be specified in the **Additional Options** box located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-dollar

The `-dollar` (dollar format) switch directs the compiler to accept dollar signs (\$) in identifiers. This option is disabled by default.

-dry

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

-dryrun

The `-dryrun` (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 standard output stream (<stdout>) 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.

-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 Analog Devices keyword extensions to ISO/ANSI standard C and C++. This recognition includes keywords such as `pm` and `dm` without leading underscores which could affect conforming ISO/ANSI C and C++ programs. This is the default mode. The `-no-extra-keywords` switch ([on page 1-27](#)) can be used to disallow support for the additional keywords. [Table 1-8 on page 1-55](#) provides a list and a brief description of keyword extensions.

-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, such as:

Table 1-6. Build Tools' Options

Option	Tool
<code>-flags-asm</code>	<code>easm219x</code> Assembler
<code>-flags-compiler</code>	Compiler executable
<code>-flags-lib</code>	<code>elfar</code> Library Builder
<code>-flags-link</code>	Linker

-full-version

The `-full-version` (display versions) switch directs the compiler to display version information for build tools used in a compilation.

Compiler Command-Line Interface

-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. When 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 only a list of the files included by the preprocessor via the `#include` directive, without compiling.

-HH

The `-HH` (list headers and compile) switch directs the compiler to output to the standard out 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.

-I-

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

 For header files in angle brackets the compiler performs the latter two searches only.

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

 This switch removes the directory containing the current input file from the `include` directory list.

-I 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 option 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 always processed before an **-include** file. Only one **-include** may be given.

Compiler Command-Line Interface

-J

The `-J` switch directs the preprocessor to perform "old-style" preprocessing.

-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 when linking. The library name is the portion of the file name between the `lib` prefix and `.dllb` extension. For example, the compiler command-line switch `-lc` directs the linker to search in the library named `c` for functions. This library resides in a file named `libc.dllb`.

Normally, you should list all object files on the command line before the `-l` switch. This ensures that the functions and global variables the object files refer to are loaded in the given order. This option 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, which is 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 `stdout` 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.

-MQ

The `-MQ` switch directs the compiler not to compile the source file but to output a rule. In addition, the `-MQ` switch does not produce any notification when input files are missing.

-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 **filename** argument; if the **filename** 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 to not accept alternative operator keywords and digraph sequences in the source files. For more information, see the `-alttok` switch [on page 1-23](#).

Compiler Command-Line Interface

-no-builtin

The `-no-builtin` (no builtin functions) switch directs the compiler to recognize only built-in functions that begin with two underscores (`__`). Note that this switch influences many functions. This switch also predefines the `__NO_BUILTIN` preprocessor macro. For more information on builtin functions, see [“Compiler Built-in Functions” on page 1-85](#).

-no-defs

The `-no-defs` (disable defaults) switch directs the preprocessor not to define any default preprocessor macros, include directories, library directories, libraries, or run-time headers. It also disables the Analog Devices compiler C/C++ keyword extensions.

-no-dir-warnings

The `-no-dir-warnings` (disable directory warning) switch directs the compiler not to issue warnings when it encounters directories on the command line that do not exist. Such directories might be used as part of subsequent `-I` (include search directory) and `-L` (library search directory) switches.

-no-dollar

The `-no-dollar` (disable dollar format) switch directs the compiler to not accept dollar signs (\$) in identifiers.

-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 ISO/ANSI programs. These extensions include keywords, such as `pm` and `dm`, which may be used as identifiers in conforming programs. The alternate keywords that are prefixed with two

leading underscores, such as `__pm` and `__dm`, continue to work. The “**-no-extra-keywords**” switch ([on page 1-32](#)) can be used to explicitly request support for the additional keywords.

-no_hardware_pc_stack

The `-no_hardware_pc_stack` switch directs the compiler to produce code to avoid the possibility of overflowing the hardware call stack. This is done by popping, from the hardware PC stack, the return address at the start of each function and storing this value on the software data stack to be pushed back on the hardware stack just before returning from the function.

-no-inline

The `-no-inline` (disable inline keyword) switch directs the compiler not to perform any high-level optimizations and directs the compiler to ignore the `inline` keyword.

-no-std-def

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

 This switch also disables the Analog Devices keyword extensions which have no leading underscores, such as `pm` and `dm`.

-no-std-inc

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

Compiler Command-Line Interface

-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 when the `-threads` (enable thread-safe build) switch is not used.

-no-widen-muls

The `-no-widen-muls` (disable widening multiplications) switch disables the compiler optimization which it performs on multiplication operations.

By default, the compiler, attempts to optimize integer multiplication operations which are stored in a long result to utilize the double word MAC result registers of the ADSP-219x DSPs. The code produced this way is better suited to the processor and therefore more efficient.

However, this optimization can generate overflow results which are not consistent in some cases and may differ from expected results depending on the optimisations enabled and the way that the source is written. The inconsistency and differences are seen if an overflow and truncation of the integer operands would normally occur.

When the optimization is applied, there is no truncation. When the optimization is disabled, the result of overflow will be truncated to integer size before being stored in the long result.

-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 `cc219x` compiler.



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

-Os

The `-Os` (optimize for size) switch directs the compiler to produce code that is optimized for size. This is achieved by performing all optimizations except those that increase code size. The optimizations not performed include loop unrolling, some delay slot filling, and jump avoidance. The compiler also uses a function to save and restore preserved registers for a function instead of generating the more cycle-efficient inline default versions.

-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 preprocessor runs (without compiling) and to omit the `#line` preprocessor directives (with line number information) in the output from the preprocessor. The `-C` switch may be used in combination with the `-P` switch.

Compiler Command-Line Interface

-PP

The `-PP` (omit line numbers and compile) switch directs the compiler to omit the `#line` preprocessor directives with line number information from the preprocessor output. After preprocessing, compilation proceeds normally.

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

The `-path` (tool location) switch directs the compiler to use the specified component in place of the default-installed version of the compilation tool. The component should comprise a relative or absolute path to its location. Respectively, the tools are the assembler, compiler, driver definitions file, librarian and linker. Use this switch when you wish to override the normal version of one or more of the tools. The `-path-tool` switch also 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.

-path-temp directory

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

-pch

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

 Precompiled header files can significantly speed compilation; precompiled headers tend to occupy more disk space.

-pchdir directory

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

-pedantic

The `-pedantic` (ANSI standard warnings) switch causes the compiler to issue warnings for any constructs found in your program that do not strictly conform to the ISO/ANSI standard for C or C++.

 The compiler may not detect all noconforming constructs. In particular, the `-pedantic` switch does not cause the compiler to issue errors when Analog Devices keyword extensions are used.

-pedantic-errors

The `-pedantic-errors` (ANSI C errors) switch causes the compiler to issue errors instead of warnings for cases described in the `-pedantic` switch.

Compiler Command-Line Interface

-pplist filename

The `-pplist` (preprocessor listing) switch 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 as:

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
R	Diagnostic message (remark)
W	Diagnostic message (warning)
E	Diagnostic message (error)
C	Diagnostic message (catastrophic error)

-proc identifier

The `-proc` (target processor) switch specifies that the compiler should produce code suitable for the identified DSP. If the processor identifier is unknown to the compiler, it will attempt to read required switches for code generation from a `<identifier>.ini` file. It will search 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 project directory. This option is position-dependent on the command line; that is, it affects only source files that follow the option.

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

-R-

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

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

-reserve *register*[, *register* ...]

The -reserve (reserve register) switch directs the compiler **not** to use the specified register(s). This guarantees that a known register or set of registers is available for autobuffering.

You can reserve the I2, I3, and M0 registers. Separate register names with commas on the compiler command line. Reserving registers seriously reduces the effectiveness of compiler optimizations and should only be done when essential.

-S

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

Compiler Command-Line Interface



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

-s

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

-save-temps

The `-save-temps` (save intermediate files) switch directs the compiler not to discard intermediate files. The compiler places the intermediate output (`*.i`, `*.is`, `*.s`) files in the current temp directory. See [Table 1-2 on page 1-10](#) 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 option 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 when the `-unsigned-char` (make char unsigned) switch is not used.

-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.

-sysdefs

The `-sysdefs` (system definitions) switch directs the compiler to define several preprocessor macros describing the current user and user's system. The macros are defined as character string constants and are used in functions with null-terminated string arguments.

The following macros will be defined if the system returns information for them:

<u>Macro</u>	<u>Description</u>
<code>__HOSTNAME__</code>	The name of the host machine
<code>__MACHINE__</code>	The machine type of the host machine
<code>__SYSTEM__</code>	The OS name of the host machine
<code>__USERNAME__</code>	The current user's login name
<code>__GROUPNAME__</code>	The current user's group name
<code>__REALNAME__</code>	The current user's real name.



`__MACHINE__`, `__GROUPNAME__`, and `__REALNAME__` are not available on Windows™ platforms.

-T filename

The `-T` (Linker Description File) switch directs that the linker, when invoked, will use the specified Linker Description File (LDF). If `-T` is not specified, a default `.LDF` file is selected based on the processor variant.

Compiler Command-Line Interface

-threads

The `-threads` (enable thread-safe build) specifies that the build and link should be thread-safe. The macro `_ADI_THREADS` is defined to one (1). It is used for conditional compilation by the preprocessor and by the default Linker Description Files to link with thread-safe libraries.



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

-time

The `-time` (time the compiler) switch directs the compiler to display the 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. The compiler processes all `-D` (define macro) switches on the command line before any `-U` (undefine macro) switches.



You can invoke this switch by selecting the **Undefines** field in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Preprocessor** category.

-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 both the version and command-line information for all the compilation tools as they process each file.

-val-global <name-list>

The `-val-global` (add global names) switch directs the compiler that the names given by **<name-list>** are present in all globally defined variables. The list is separated by double colons(`::`). In C++, these names are encoded as enclosing namespace or classes. In C, the names are prefixed and separated by underscores (`_`). The compiler will issue an error on any globally defined variable in the current source module(s) not using **<name-list>**. This switch is used to define VCSE components.

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

-W {error|remark|suppress|warn} number

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

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

Compiler Command-Line Interface

-Wdriver-limit number

The `-Wdriver-limit` (maximum process errors) switch lets you set a maximum number of driver errors (command line, etc.) that the driver aborts at.

-Werror-limit number

The `-Werror-limit` (maximum compiler errors) switch lets you set 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.

-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.

-warn-protos

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

-workaround <workaround>[,<workaround>]*

The `-workaround` switch enables code generator workaround for specific hardware defects. Example of a valid workaround: `type32a-anomaly`.

-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 with handling long file names, which can make the compiler driver's command line too long for some operating systems.

-write-opts

The `-write-opts` switch directs the compiler to pass the user-specified options (but **not** the input file names) to the main driver via a temporary file. This can be helpful if the resulting main driver command line would otherwise be too long.

-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

Compiler Command-Line Interface

is written to the named file. `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++.

-bool

The `-bool` (accept boolean) switch directs the compiler to predefine the new `bool` keyword and the `true` and `false` constants. The compiler also defines the `_BOOL` preprocessor macro when this option (or another option which directs the compiler to recognize `bool`) is given. This switch is enabled by default in C++ mode.

-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.

-force

The `-force` (force definitions) switch directs the compiler to force the virtual function table definitions in cases where the heuristic used by the compiler to determine the definition provides no guidance.

The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline, non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table as a local static entity. This option can be used to force the definition of a virtual function table for such classes by overriding the described behavior.

-instant{all|local|used}

The default behavior that the compiler uses to perform template instantiation is to suppress the instantiation of any templates on the first compilation and let the prelinker compiler utility decide which files need to be recompiled to instantiate the required templates. This default behavior may be modified by the use of the `-instantused`, `-instantlocal` and `-instantall` switches.

The `-instantused` (instantiate all used templates) switch causes the compiler to instantiate any template entities that are seen to be used when as part of performing the first compilation.

The `-instantlocal` (instantiate used with internal linkage) switch is similar to `-instantused` except that all templates are given internal linkage.

The `-instantall` (instantiate all templates) switch directs the compiler to instantiates all template entities as part of the first compilation. The instantiation of template entities is performed whether they are used or not when this switch is used.

These switches will not conflict with normal use of the prelinker.

Compiler Command-Line Interface

-namespace

The `-namespace` (namespace) switch directs the compiler to enable support for namespaces. This is the default mode when the `-no-namespace` (disable namespace) switch is not used.

-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 this switch is used. This is the default mode.

-no-bool

The `-no-bool` (disable boolean) switch directs the compiler to disable the `bool` keyword.



The compiler supports the `bool` data type in C++ mode by default. For more information, see [on page 1-46](#).

-no-demangle

The `-no-demangle` (disable demangler) switch directs the compiler to prevent the driver from filtering any 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 [on page 1-46](#).

-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 [on page 1-48](#).

-no-restrict

The `-no-restrict` (disable restrict) switch directs the compiler to disable recognition of the `restrict` keyword as a type qualifier for pointers and function parameter arrays that decay to pointers.

-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 new `wchar_t` construct.

Compiler Command-Line Interface

-restrict

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

-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. Both `-strict` and `-ansi-c++` define 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. Both `-strictwarn` and `-ansi-c++` define the `__STRICT_ANSI__` macro.

-suppress

The `-suppress` (`suppress`) switch directs the compiler to suppress the virtual function table definitions in cases where the heuristic used by the compiler to determine the definition provides no guidance.

The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline, non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table as a local static entity. This option can be used to suppress the definition of a virtual function table for such classes by overriding the described behavior.

-tpautooff

The `-tpautooff` (disable automatic template instantiation) switch directs the compiler to disable automatic instantiation of templates and 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 new `wchar_t` construct. This is the default setting when the `-no-wchar` (disable wide char type) switch is not used.

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 1-7](#) shows the size used for each of the intrinsic C/C++ data types

Table 1-7. Data Type Sizes for the ADSP-219x DSPs

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

On any platform the basic type `int` will be the native word size. The data type `long` is 32 bits, as is `float`. A `pointer` is the same size as an `int`.

In the ADSP-219x processor 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, `double` word data types should be expected to run more slowly, relying largely on software-emulated arithmetic.

Analog Devices does not support data sizes smaller than a single word location for the ADSP-219x processors. For the current processors, this means that both `short` and `char` have the same size as `int`. Although 16-bit `chars` are unusual, they do conform to the standard.

Type `double` poses a special problem. The C language tends to default to `double` for constants and in many floating-point calculations. Without some special handling, many programs would inadvertently end up using slow-speed emulated 64-bit floating-point arithmetic, even when variables are declared consistently as `float`.

In order to avoid this problem and provide the best performance, the size of `double` on the ADSP-219x DSPs is always 32 bits. This should be acceptable for most DSP programming. It is not, however, fully standard conforming.

The standard `include` files automatically redefine the math library interfaces such that functions like `sin` can be directly called with the proper size operands. Therefore,

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

For full descriptions of these functions and their implementation, see Chapter 3, “[DSP Run-Time Library](#)”.

C/C++ Compiler Language Extensions

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

This section contains:

- [“Inline Function Support Keyword \(inline\)” on page 1-57](#)
- [“Inline Assembly Language Support Keyword \(asm\)” on page 1-58](#)
- [“Dual Memory Support Keywords \(pm dm\)” on page 1-69](#)
- [“Placement Support Keyword \(section\)” on page 1-74](#)
- [“Boolean Type Support Keywords \(bool, true, false\)” on page 1-75](#)
- [“Pointer Class Support Keyword \(restrict\)” on page 1-75](#)
- [“Variable Length Array Support” on page 1-76](#)
- [“Non-Constant Aggregate Initializer Support” on page 1-78](#)
- [“Indexed Initializer Support” on page 1-78](#)
- [“Aggregate Constructor Expression Support” on page 1-81](#)
- [“Fractional Type Support” on page 1-82](#)
- [“Preprocessor Generated Warnings” on page 1-85](#)
- [“C++ Style Comments” on page 1-85](#)
- [“Compiler Built-in Functions” on page 1-85](#)
- [“ETSI Support” on page 1-91](#)
- [“Pragmas” on page 1-99](#)

The additional keywords that are part of these C/C++ extensions do not conflict with any ISO/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 forms of the keyword extension that omits the leading underscores. See [“-extra-keywords” on page 1-27](#) for more information.

This section describes only the shorter forms of the keyword extensions, but 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 use `inline` or `__inline` interchangeably in your code.

You might need to use the longer forms (such as `__inline`) exclusively if you are porting a program that uses the extra Analog Devices keywords as identifiers. For example, a program might declare local variables such as `pm` or `dm`. In this case, you should use the `-no-extra-keywords` switch, and if you need to declare a function as `inline`, or allocate variables to memory spaces, you can use `__inline` or `__pm/__dm` respectively.

[Table 1-8](#) provides a list and a brief description of keyword extensions. [Table 1-9](#) provides a list and a brief description of operational extensions. Both tables direct you to sections of this chapter that document each extension in more detail.

Table 1-8. Keyword Extensions

Keyword extensions	Description
<code>inline</code> (function)	Directs the compiler to integrate the function code into the code of the callers. For more information, see “Inline Function Support Keyword (inline)” on page 1-57.
<code>dm</code>	Specifies the location of a static or global variable or qualifies a pointer declaration “*” as referring to Data Memory. For more information, see “Dual Memory Support Keywords (pm dm)” on page 1-69.

C/C++ Compiler Language Extensions

Table 1-8. Keyword Extensions (Cont'd)

Keyword extensions	Description
pm	Specifies the location of a static or global variable or qualifies a pointer declaration “*” as referring to Program Memory. For more information, see “Dual Memory Support Keywords (pm dm)” on page 1-69.
section(“string”)	Specifies the section in which an object or function is placed. For more information, see “Placement Support Keyword (section)” on page 1-74.
bool, true, false	A Boolean type. For more information, see “Boolean Type Support Keywords (bool, true, false)” on page 1-75.
restrict keyword	Specifies restricted pointer features. For more information, see “Pointer Class Support Keyword (restrict)” on page 1-75.

Table 1-9. Operational Extensions

Operation extensions	Description
Variable length arrays	Support for variable length arrays lets you use automatic arrays whose length is not known until runtime. For more information, see “Variable Length Array Support” on page 1-76.
Non-constant initializers	Support for 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 1-78.
Indexed initializers	Support for indexed initializers lets you specify elements of an aggregate initializer in an arbitrary order. For more information, see “Indexed Initializer Support” on page 1-78.
Preprocessor generated warnings	Support for generating warning messages from the preprocessor. For more information, see “Preprocessor Generated Warnings” on page 1-85.
C++-style comments	Support for C++-style comments in C programs. For more information, see “C++ Style Comments” on page 1-85.

Inline Function Support Keyword (inline)

The `inline` keyword directs cc219x to integrate the code for the function you declare as `inline` into the code of its callers. Using this keyword eliminates the function-call overhead and therefore can increase the speed of your program's execution. Argument values that are constant and that have known values may permit simplifications at compile time.

The following example shows a function definition that uses the `inline` keyword.

```
inline int single_addition (int *a)
{
    (*a)++;
}
```

A function declared `inline` must be defined (its body must be included) in every file in which the function is used. The normal way to do this is to place 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, the address is not needed for an `inline` function called only 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` (on page 1-33) and `-traditional` (on page 1-21) switches disable function inlining.
-  If creating a debug target (or using the `-g` switch on the command line) and optimizations are not enabled, the compiler ignores the `inline` keyword and disables function inlining. This process allows users maximum debug capabilities.

Inline Assembly Language Support Keyword (`asm`)

The cc219x `asm()` construct lets you code ADSP-219x assembly language instructions within a C or C++ function. The `asm()` construct is useful for expressing assembly language statements that cannot be expressed easily or efficiently with C constructs.

The `asm()` keyword allows you code complete assembly language instructions or you can specify the operands of the instruction using C expressions. When specifying operands with a C expression, you do not need to know which registers or memory locations contain C variables.

The C 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* through to the assembler without reading or analyzing it.

 Any `asm()` constructs defined before the variable declarations within `main()` are flagged as errors, because executable statements are not allowed before declarations in C code.

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

```
asm(" ENA INT;");
```

The complete assembly language instruction, enclosed in quotes, is the argument to `asm()`.

 The compiler generates a label before and after inline assembly instructions when generating debug code (`-g` switch). These labels are used to generate the debug line information used by the debugger. If the inline assembler inserts conditionally assembled code, an undefined symbol error is likely to occur at link time. If the inline assembler changes the current section and thereby causes the compiler labels to be placed in another section, such as a data section (instead of the default code section), then the debug line information will be incorrect for these lines.

Using `asm()` constructs with operands requires some additional syntax. 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 1-65](#)
- [“Assembly Construct Reordering and Optimization” on page 1-66](#)
- [“Assembly Constructs with Input and Output Operands” on page 1-67](#)
- [“Assembly Constructs and Macros” on page 1-68](#)

Assembly Construct Template

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

ASM() Construct 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:

- *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 appearance from left to right, starting at 0. Separate multiple instructions with a semicolon, and enclose the entire string within double quotes. For more information on templates containing multiple instructions, see [“Assembly Constructs with Multiple Instructions” on page 1-65](#).

- *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 variable that receives output from a corresponding operand in the assembly instruction.

- *input operand*

The input operand is a 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 over-written by the assembly instructions. Use lowercase characters to name clobbered registers. Enclose each register name within double quotes, and separate the quoted register names with commas.

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 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, and the last input operand from the clobbered registers. If there are no output operands and there are input operands, there must be two consecutive colons separating the assembly template from the input operands.
- 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 ADSP-219x assembly language `abs` instruction:

```
{
    int x, result;

    asm ("%0=abs %1;" : "=c" (result) : "c" (x));
}
```

In the previous example, note the following points:

- The template is `"%0=abs %1;"`. The `%0` is replaced with operand zero (`result`), the first operand. The `%1` is replaced with operand one (`x`).
- The output operand is the C variable: `result`. The letter `c` is the *operand constraint* for the variable. This constrains the output to an ALU result register. The compiler generates code to copy the output from the register to the variable `result`, if necessary. The “=” in `=c` indicates that the operand is an output.
- The input operand is the C variable `x`. The letter `c` is the *operand constraint* for the variable. This constrains `x` to an ALU register. If `x` is stored in different kinds of registers or in memory, the compiler generates code to copy the values into an 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. There are several pieces of information that need to be conveyed for the compiler to know how to assign registers to operands. This information is conveyed with an oper-

and 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 which describes the class of allowable registers. [Table 1-10 on page 1-64](#) describes the correspondence between constraint letters and register classes.

 The use of any letter not listed in [Table 1-10](#) results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

For example, if your assembly template contains “`ax1 = dm(%0
+= m3);`” and the address you want to load from is in the variable `p`, the compiler needs to know that it should put `p` in a DAG1 I register (I0–I3) before it generates your instruction. You convey this information to `cc219x` by specifying the operand “`w`” (`p`) where “`w`” is the constraint letter for DAG1 I registers.

To assign registers to the operands, the compiler must also be told 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 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 `=`.

C/C++ Compiler Language Extensions

- The compiler may allocate an output operand in the same register as an unrelated input operand, unless the output operand has the `&=` constraint modifier. This situation can occur because the compiler assumes that 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. See [Table 1-10](#) for the list of operand constraints.

Table 1-10. ASM() Operand Constraints

Constraint ¹	Description	Registers
b	input xregs to MAC	MX1, MX0, SR1, SR0, MR1, MR0, AR
B	input yregs to MAC	MY1, MY0
c	results from ALU	AR
C	result from int multiplies	MRO
cc	Used in the clobber list to tell the compiler that condition codes have been clobbered	ASTAT
d	input xregs to SHIFTER	SI, SR1, SR0, MR1, MR0, AX0, AY0, AX1, AY1, MX0, MX1, MY0, MY1, AR
D	result from shift	SR1
e	data registers, size 16	SI, AX1, AX0, MX1, MX0, MY0, MY1, AY1, AY0, MR1, MR0, SR1, SR0, AR
f	shift amount	SE
g	ALU X registers	AX1 AX0 AR SR1 SR0 MR1 MR0
G	ALU Y registers	AY1 AY0

Table 1-10. ASM() Operand Constraints (Cont'd)

Constraint ¹	Description	Registers
memory	Used in the clobber list to tell the compiler that the asm() statement writes to memory	
r	all registers	SR1, SR0, SI, MY1, MX1, AY1, AX1, MY0, MX0, AY0, AX0, MR1, MR0, AR, I0-I7, M0-M7, L0-L7
u	DAG1 L registers	L0-L3
v	DAG2 L registers	L4-L7
w	DAG1 I registers	I0-I3
x	DAG1 M registers	M0-M3
y	DAG2 I registers	I4-I7
z	DAG2 M registers	M4-M7
=&constraint	Indicates that the constraint is applied to an output operand that may not overlap an input operand	
=constraint	Indicates that the constraint is applied to an output operand	

¹ The use of any letter not listed in [Table 1-10](#) results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

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 or by quoting each line separately.

This is an example of multiple instructions in a template:

```
asm ("se=exp %1 (hi);"  
    "sr=norm %1 (hi);"  
    "%0=sr0;"  
    : "=e" (normalized)      // output  
    : "e" (inval) ;          // input  
    : "se", "sr1", "sr0") ; // 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. This assumption does not mean that you cannot use instructions with side effects, but you must be careful. The compiler may eliminate them if the output operands are not used, or move them out of loops, or replace two with one if they constitute a common subexpression.

Also, if your instruction does have 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 I0write(val,addr) \  
asm volatile ("si="#val";I0("#addr")=si;": : : "si");
```

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 only one `asm volatile()` construct, or use the output of the `asm()` construct in a C statement.

Assembly Constructs with Input and Output Operands

The assembly constructs' output operands must be write only; cc219x assumes that the values in these operands do not need to be preserved. When the assembler instruction has an operand that is both 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 them is expressed by constraints that say they need to be in the same location when the instruction executes.

You can use the same C expression for both operands, or different expressions. For example, in the following statement, the `modify` instruction uses `sock` as its read only source operand and `shoe` as its read-write destination:

```
/* (pseudo code) modify (shoe += sock); */
asm ("modify (%0 += %2);":"=w"(shoe):"0"(shoe),"x"(sock));
```

The constraint `"0"` for operand 1 says that it must occupy the same location as operand 0. A digit in an operand constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand is in the same place as another operand. Just because a variable (for example `shoe` in the code that follows) is used for more than one operand does not guarantee that the operands are in the same place in the generated assembler code.

```
/* Do NOT try to control placement with operand names; use
the % digit. The following code might NOT work.*/
asm ("modify (%0 += %2);":"=w"(shoe):"w"(shoe),"x"(sock));
```

In some cases, operands 0 and 1 could be stored in different registers due to reloading or optimizations.

Be aware that `asm()` does not support input operands that are used as both read operands and write operands. The example below shows a *dangerous* use of such an operand. In this example, `my_variable` is modified during

C/C++ Compiler Language Extensions

the `asm()` operation. The compiler only knows that the output, `result_asm`, has changed. Subsequent use of `my_variable` after the `asm()` instruction may yield incorrect results since those values may have been modified during the `asm()` instruction and may not have been restored.

```
int result_asm;
int *my_variable;
/* NOT recommended */
/* (pseudo code) result_asm = dm(*my_variable += 3); */
/* asm() operation changes value of my_variable */

asm("%0=DM(%1 += 3);":"=e"(result_asm):"w"(my_variable));
```

Assembly Constructs and Macros

One way to use `asm()` constructs is to encapsulate them in macros that look like functions. For example, the following shows macros that contain `asm()` constructs. This code defines a macro, `abs_macro()`, which uses the `inline asm()` instruction to perform an assembly-language abs operation of variable `x_var`, putting the result in `result_var`:

```
#define abs_macro(result,x) \
asm("%0=abs %1;":"=c" (result):"c"(x))

/* (pseudo code) result = abs x */

main(){
    int result_var=0;
    int x_var=10;

    abs_macro(result_var, 10);
    /* or */
    abs_macro(result_var, x_var);
}
```

Dual Memory Support Keywords (pm dm)

This section describes `cc219x` keyword extensions to C and C++ that support the dual-memory space, modified Harvard architecture of the ADSP-219x processors. There are two keywords used to designate memory space: `dm` and `pm`. These keywords can be used to specify the location of a static or global variable or to qualify a pointer declaration.

These keywords allow you to control placement of data in primary (`dm`) or secondary (`pm`) data memory. No data is placed in the memory unit that holds programs. The following rules apply to dual memory support keywords:

- A memory space keyword (`dm` or `pm`) refers to the expression to its right.
- You can specify a memory space for each level of pointer. This corresponds to one memory space for each `*` in the declaration.
- The compiler uses Data Memory as the default memory space for all variables. All undeclared spaces for data are Data Memory spaces.
- The compiler always uses Program Memory as the memory space for functions. Function pointers always point to Program Memory.
- You cannot assign memory spaces to automatic variables. All automatic variables reside on the stack, which is always in Data Memory.
- Literal character strings always reside in Data Memory.
- Although program memory on the ADSP-219x DSPs consists of 24-bit words, only 16 bits of each word are used when C or C++ data is stored in `pm`. (This is normally the case for assembly lan-

C/C++ Compiler Language Extensions

guage programming as well.) If you need special access to all 24 bits, you should use an assembly language subroutine and work with the PX register.

The following listing shows examples of dual memory keyword syntax.

```
int pm abc[100];
/* declares an array abc with 100 elements in Program Memory */
int dm def[100];
/* declares an array def with 100 elements in Data Memory */
int ghi[100];
/* declares an array ghi with 100 elements in Data Memory */
int pm * pm pp;
/* declares pp to be a pointer which resides in Program Memory
   and points to a Program Memory integer */
int dm * dm dd;
/* declares dd to be a pointer which resides in primary Data
   Memory and points to a Data Memory integer */
int *dd;
/* declares dd to be a pointer which resides in Data Memory
   and points to a Data Memory integer */
int pm * dm dp;
/* declares dp to be a pointer which resides in Data Memory
   and points to a Program Memory integer */
int pm * dp;
/* declares dp to be a pointer which resides in Data Memory
   and points to a Program Memory integer */
int dm * pm pd;
/* declares pd to be a pointer which resides in pm (secondary
   Data Memory) and points to a Data Memory integer */
int * pm pd;
/* declares pd to be a pointer which resides in Program memory
   and points to a Data Memory integer */
float pm * dm * pm fp;
/* the first pm means that *fp is in Program Memory,
```

```

the following dm puts *fp in Data Memory, and fp
itself is in Program Memory */

```

Memory space specification keywords cannot qualify type names and structure tags, but you can use them in pointer declarations. The following listing shows examples of memory space specification keywords in typedef and struct statements.

```

/* Dual Memory Support Keyword typedef & struct Examples */
typedef float pm * PFL0ATP;
/* PFL0ATP defines a type which is a pointer to a */
/* float which resides in pm. */

struct s {int x; int y; int z;};
static pm struct s mystruct={10,9,8};
/* Note that the pm specification is not used in */
/* the structure definition. The pm specification */
/* is used when defining the variable mystruct */

```

Memory Keywords and Assignments/Type Conversions

Memory space specifications limit the kinds of assignments your program can make:

- You may make assignments between variables allocated in different memory spaces.
- Pointers to program memory must always point to PM. Pointers to data memory must always point to DM. You may not mix addresses from different memory spaces within one expression. Do not attempt to explicitly cast one type of pointer to another.

C/C++ Compiler Language Extensions

The following listings show a code segment with variables in different memory spaces being assigned and a code segment with illegal mixing of memory space assignments.

```
/* Legal Dual Memory Space Variable Assignment Example */
int pm x;
int dm y;
x = y; /* Legal code */
```

```
/* Illegal Dual Memory Space Type Cast Example */
int pm *x;
int dm *y;
int dm a;
x = y; /* Compiler will flag error */
x = &a; /* Compiler will flag error */
```

Memory Keywords and Function Declarations/Pointers

Functions always reside in program memory. Pointers to functions always point to PM. The following listing shows some example function declarations with pointers.

```
/* Dual Memory Support Keyword Function Declaration (With Pointers) Syntax Examples */
```

```
int * y();      /* function y resides in */
                /* pm and returns a      */
                /* pointer to an integer */
                /* which resides in dm   */
```

```
int pm * y();   /* function y resides in */
                /* pm and returns a      */
                /* pointer to an integer */
                /* which resides in pm   */
```

```

int dm * y();    /* function y resides in */
                /* pm and returns a      */
                /* pointer to an integer */
                /* which resides in dm   */

int * pm * y(); /* function y resides in */
                /* pm and returns a      */
                /* pointer to a pointer  */
                /* residing in pm that   */
                /* points to an integer  */
                /* which resides in dm   */

```

Memory Keywords and Function Arguments

The compiler checks calls to prototyped functions for memory space specifications consistent with the function prototype. The following example shows sample code that cc219x flags as inconsistent use of memory spaces between a function prototype and a call to the function.

```

/* Illegal Dual Memory Support Keywords & Calls To Prototyped
Functions */

extern int foo(int pm*);
/* declare function foo() which expects a pointer to an int
residing in pm as its argument and which returns an int */

int x;    /* define int x in dm */

foo(&x); /* call function foo()
        /* using pm pointer (location of x) as the
        /* argument. cc219x FLAGS AS AN ERROR; this is an
        /* inconsistency between the function's
        /* declared memory space argument and function
        /* call memory space argument

```

Memory Keywords and Macros

Using macros when making memory space specification for variables or pointers can make your code easier to maintain. If you must change the definition of a variable or pointer (moving it to another memory space), declarations that depend on the definition may need to be changed to ensure consistency between different declarations of the same variable or pointer.

To make changes of this kind easier, you can use C preprocessor macros to define common memory spaces that must be coordinated. The following listing shows two code segments that are equivalent after preprocessing.

The following code segment demonstrates how you can redefine the memory space specifications by redefining the macros `SPACE1` and `SPACE2`.

```
/* Dual Memory Support Keywords & Macros */
#define SPACE1 pm
#define SPACE2 dm

char pm * foo (char dm *)    char SPACE1 * foo (char SPACE2 *)
char pm *x;    char SPACE1 *x;
char dm y;     char SPACE2 y;

x = foo(&y);    x = foo(&y);
```

Placement Support Keyword (section)

The `section` keyword directs the compiler to place an object or function in an assembly `.SECTION`, in the compiler's intermediate assembly output file. You name the assembly `.SECTION` with `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 sections.

Applying `section()` is only meaningful 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, i.e. are explicitly `static`, or are given as external-object definitions. The example 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 to ANSI C that support the C++ Boolean type. The `bool` keyword is a unique signed integral type. There are two built-in constants of this type: `true` and `false`. When converting a numeric or pointer value to `bool`, a zero value becomes `false`; a nonzero value becomes `true`. A `bool` value may be converted to `int` by promotion, taking `true` to one and `false` to zero. A numeric or pointer value is automatically converted to `bool` when needed.

These keywords behave more or less as if the declaration that follows had appeared at the beginning of the file, except that assigning a nonzero 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` operator 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 that you can identify that a pointer does not create an alias. Also, two different restricted pointers can not designate the same object and therefore are not

aliases. The compiler is free to use the information about restricted pointers and aliasing in order 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 fi (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 except for the following cases:

- 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 you have a program that uses a restricted pointer in a way that it does not uniquely refer to storage, then the behavior of the program is undefined.

Variable Length Array Support

The compiler supports variable-length automatic arrays. Unlike other automatic arrays, variable-length ones are declared with a non-constant length. This means that the space is allocated when the array is declared, and deallocated when the brace-level is exited.

The compiler does not allow jumping into the brace-level of the array and produces a compile time error message if this is attempted. The compiler does allow breaking or jumping out of the brace-level, and it deallocates the array when this occurs.

You can use variable-length arrays as function arguments, as shown in the following example.

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

The compiler calculates the length of an array at the time of allocation. It then remembers the array length until the brace-level is exited and can return it as the result of the `sizeof()` function performed on the array.

Because variable length arrays must be stored on the stack, it is impossible to have variable length arrays in Program Memory (pm). The compiler issues an error if an attempt is made to use a variable length array in pm.

As an example, if you were to implement a routine for computation of a product of three matrices, you need to allocate a temporary matrix of the same size as input matrices. Declaring automatic variable size matrix is much easier then explicitly allocating it in a heap.

The expression declares an array with a size that is computed at run time. The length of the array is computed on entry to the block and saved in case `sizeof()` is applied to the array. For multidimensional arrays, the boundaries are also saved for address computation. After leaving the block all the space allocated for the array and size information will be deallocated.

For example, the following program prints 40, not 50:

```
#include <stdio.h>
void foo(int);

main ()
{
    foo(40);
```

```
}

void foo (int n)
()
{
    char c[n];
    n = 50;
    printf("%d", sizeof(c));
}
```

Non-Constant Aggregate Initializer Support

The cc219x compiler includes support for the ISO/ANSI standard definition of C and C++ and also includes extended support for 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 };
}
```

Indexed Initializer Support

ISO/ANSI Standard C and C++ requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized. The cc219x compiler, by comparison, supports labeling elements for array initializers. This feature lets you specify

array or structure elements in any order by specifying the array indices or structure field names to which they apply. All index values must be constant expressions, even in automatic arrays.

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 initializer is in ISO/ANSI standard C/C++; the second initializer uses the cc219x compiler.



The `[index]` precedes the value being assigned to that element.

```
/* Example 1 Standard & cc219x C/C++ Array Initializer */
/* Standard Array Initializer */

int a[6] = { 0, 0, 115, 0, 29, 0 };

/* equivalent cc219x C/C++ array initializer */

int a[6] = { [2] 115, [4] 29 };
```

You can combine this technique of naming elements with standard C/C++ initialization of successive elements. The standard and cc219x instructions below are equivalent. Note that any unlabeled initial value is assigned to the next consecutive element of the structure or array.

```
/* Example 2 Standard & cc219x C/C++ Array Initializer */
/* Standard Array Initializer */

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

C/C++ Compiler Language Extensions

```
/* equivalent cc219x C/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 Array Initializer With enum Type Indices */
/* cc219x C/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 a field to initialize with *field name* before the element value. The standard C/C++ and cc219x C/C++ struct initializers in the example below are equivalent.

```
/* Example 4 Standard & cc219x C/C++ struct Initializer */
/* Standard struct Initializer */

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

/* Equivalent cc219x C/C++ struct Initializer With
Labeled Elements */

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

Aggregate Constructor Expression Support

Extended initializer support in cc219x C/C++ includes support for aggregate constructor expressions. These expressions enable you to assign values to large structure types without requiring each element's value to be individually assigned.

The following example shows an ISO/ANSI standard `struct` usage followed by equivalent cc219x code that has been simplified using an constructor expression.

```
/* Standard struct & cc219x C/C++ Constructor struct */
/* Standard struct */

struct foo {int a; char b[2];};
struct foo make_foo(int x, char *s)
{
    struct foo temp;
    temp.a = x;
    temp.b[0] = s[0];
    if (s[0] != '\0')
        temp.b[1] = s[1];
    else
        temp.b[1] = '\0';
    return temp;
}

/* Equivalent cc219x C/C++ constructor struct */

struct foo make_foo(int x, char *s)
{
    return((struct foo) {x, {s[0], s[0] ? s[1] : '\0'}});
}
```

Fractional Type Support

While in C++ mode, the cc219x compiler supports fractional (fixed-point) arithmetic that provides a way of computing with non-integral values within the confines of the fixed-point representation. Hardware support for the 16-bit fractional arithmetic is available on the ADSP-219x DSPs.

Fractional values are declared with the `fract` data type. Ensure that your program includes the `fract` header file. `fract` is a C++ class that supports a set of standard arithmetic operators used in arithmetic expressions. Fractional values are represented as signed values in a range of $[-1 \dots 1]$ with a binary point immediately after the sign bit. Other value ranges are obtained by scaling or shifting. In addition to the arithmetic, assignment, and shift operations, `fract` provides several type-conversion operations.

For more information about supported fractional arithmetic operators, see [“Fractional Arithmetic Operations” on page 1-83](#). For sample programs demonstrating the use of the `fract` type, see [Listing 1-1 on page 1-130](#), [Listing 1-2 on page 1-131](#), and [Listing 1-3 on page 1-131](#).



The current release of the software does not provide for automatic scaling of fractional values.

Format of Fractional Literals

Fractional literals use the floating-point representation with an “r” suffix to distinguish them from floating-point literals, for example, `0.5r`. The cc219x compiler validates fractional literal values at run time to ensure they reside within the valid range of values.

Fractional literals are written with the “r” suffix to avoid certain precision loss. Literals without an “r” are of the type `double`, and are implicitly converted to `fract` as needed.

Conversions Involving Fractional Values

The following notes apply to type-conversion operations:

- Conversion between a fractional value and a floating value is supported. The conversion to the floating-point type may result in some precision loss.
- Conversion between a fractional value and an integer value is supported. The conversion is not recommended because the only common values are 0 and -1.
- Conversion between a fractional value and a long double value is supported via `float` and may result in some precision loss.

Fractional Arithmetic Operations

The following notes summarize information about fractional arithmetic operators supported by the `cc219x` compiler:

- Standard arithmetic operations on two `fract` items include addition, subtraction, and multiplication.
- Assignment operations include `+=`, `-=`, and `*=`.
- Shift operations include left and right shifts. A left shift is implemented as a logical shift and a right shift is an arithmetic shift. Shifting left by a negative amount is not recommended.
- Comparison operations are supported between two `fract` items.
- Mixed-mode arithmetic has a preference for `fract`. For more information about the mixed-mode arithmetic, see [on page 1-84](#).

C/C++ Compiler Language Extensions

- Multiplication of a fractional and an integer produces an integer result or a fractional result. The program context determines which variant is generated following the conversion algorithm of C++. When the compiler does not have enough context, it generates an ambiguous operator message. For example,

```
error:more than one operator "*" matches these operands:
```

You cast the result of the multiply operation if the error occurs.

Mixed Mode Operations

Most operations supported for fractional values, are supported for mixed fractional/float or fractional/double arithmetic expressions. At run time, a floating-point value is converted to a fractional value, and the operation is completed using fractional arithmetic.

The assignment operations, such as `+=`, are the exception to the rule. The logic of an assignment operation is defined by the type of a variable positioned on the left side of the expression.

Floating-point operations require an explicit cast of a fractional value to the desired floating type.

Saturated Arithmetic

The `cc219x` compiler supports saturated arithmetic for fractional data in the saturated arithmetic mode.

Whenever a calculation results in a bigger value than the `fract` data type represents, the result is truncated (wrapped around). An overflow flag is set to warn the program that the value has exceeded its limits. To prevent the overflow and to get the result as the maximum representable value when processing signal data, use saturated arithmetic. Saturated arithmetic forces an overflowed value to become the maximum representable value.

The mode is set to be saturated or default with the `set_saturate_mode()` and `reset_saturate_mode()` functions. Each arithmetic operator has its corresponding variant effected in the saturated mode. For example, `add_sat`, `sub_sat`, `neg_sat`, ...

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.

C++ Style Comments

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

```
a = b
/* highly unusual */ c
```

which a standard C compiler processes as:

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

Compiler Built-in Functions

The compiler supports intrinsic functions that enable efficient use of hardware resources. Knowledge of these functions is built into the `cc219x` 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.

C/C++ Compiler Language Extensions

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` switch is used ([on page 1-32](#)).

The `cc219x` compiler provides built-in versions of some of the C library functions as described in [“Using the Compiler’s Built-In C Library Functions” on page 2-4](#).

The `sysreg.h` header file defines a set of functions that provide efficient system access to registers, modes and addresses not normally accessible from C source. These functions are specific to individual architectures and this section lists the built-in functions supported at this time on ADSP-219x DSPs.

The compiler supports:

- [“Access to System Registers” on page 1-86](#)
- [“I/O Space Read or Write” on page 1-88](#)
- [“Interrupt Control” on page 1-89](#)
- [“Mode Control” on page 1-90](#)
- [“External Memory Read or Write” on page 1-90](#)

Access to System Registers

The inclusion of `sysreg.h` allows the use of functions that will generate efficient inline instructions to implement read and write of values from and to general register set and system control set system registers.

General Register set:

ASTAT SSTAT MSTAT ICNTL IMASK IRPTL DMPG1 DMPG2 IOPG

System Control Register set:

B0 B1 B2 B3 B4 B5 B6 B7 SYSCTL CACTL

Also any 8-bit value used as a register identifier.

The prototypes for these functions are, as defined in `sysreg.h`:

```
void sysreg_write(const int sysreg, const int value);
int sysreg_read(const int sysreg);
```

The `sysreg` parameter for these functions can be a member of the `SysReg` enumeration defined in `sysreg.h`. This enumeration is used to map the actual registers to a small constant defined as a user friendly name.

The `SysReg` enumeration has the following definitions:

```
/* General Register set */
sysreg_ATEST = 0x0, // ATEST register - arithmetic status
sysreg_SSTAT = 0x1, // SSTAT register - shifter status
sysreg_MSTAT = 0x2, // MSTAT register - multiplier status
sysreg_ICNTL = 0x3, // ICNTL register - interrupt control
sysreg_IMASK = 0x4, // IMASK register - interrupts enabled mask
sysreg_IRPTL = 0x5, // Interrupt Latch register
sysreg_DMPG1 = 0x6, // DMPG1 high address register
sysreg_DMPG2 = 0x7, // DMPG2 high address register
sysreg_IOPG = 0x8, // IOPG I/O page register
/* System Control Register set */
sysreg_B0 = 0x9, // B0 base register
sysreg_B1 = 0xa, // B1 base register
sysreg_B2 = 0xb, // B2 base register
sysreg_B3 = 0xc, // B3 base register
sysreg_B4 = 0xd, // B4 base register
sysreg_B5 = 0xe, // B5 base register
sysreg_B6 = 0xf, // B6 base register
sysreg_B7 = 0x10, // B7 base register
sysreg_SYSCTL = 0x11, // SYSCTL register
sysreg_CACTL = 0x12, // Cache Control register
```

C/C++ Compiler Language Extensions

The `sysreg` parameter can also be any 8-bit value used to represent a system control register, since each variant of the ADSP-219x DSPs may have a differing System Control Register set. The compiler does not validate the `sysreg` parameter, instead it relies on the assembler to fault erroneous values.

An example use of `sysreg_read` to get the value of `IMASK` might be:

```
#include <sysreg.h>
int read_imask(){
    int value = sysreg_read(sysreg_IMASK);
    return value;
}
```

An example use of `sysreg_write` to set the value of `IMASK` might be:

```
#include <sysreg.h>
void write_imask(int val8bit) {
    sysreg_write(sysreg_IMASK, val8bit);
}
```

I/O Space Read or Write

The inclusion of `sysreg.h` allows the use of functions that will generate efficient inline instructions to implement read and write of values from and to I/O space addresses.

The prototypes for these functions are, as defined in `sysreg.h`:

```
void io_space_write(const unsigned int, const unsigned int);
int io_space_read(const unsigned int addr);
```

The `addr` parameter should be an 8- or 10-bit value that forms a 16-bit address with the use of the `IOPG` register. The compiler does not validate the `addr` parameter. If the parameter is not valid the assembler returns an error.

An example use of `io_space_read` to read from address zero might be:

```
#include <sysreg.h>
int read_io_zero(){
    int value = io_space_read(0);
    return value;
}
```

An example use of `io_space_write` to write to address zero might be:

```
#include <sysreg.h>
void write_io_zero(int val) {
    io_space_write(0, val);
}
```

Interrupt Control

The inclusion of `sysreg.h` allows the use of functions that generate the instructions to enable and disable interrupts.

The prototypes for these functions are, as defined in `sysreg.h`:

```
void enable_interrupts(void);
void disable_interrupts(void);
```

The following code provides an example of the use of `enable_interrupts` and `disable_interrupts` to disable and enable interrupts around a call to `printf`:

```
#include <sysreg.h>
#include <stdio.h>
void interrupt_safe_iprint(int val) {
    disable_interrupts();
    printf("%d\n",val);
    enable_interrupts();
}
```

Mode Control

The inclusion of `sysreg.h` allows the use of a function that generates the instructions to enable and disable a series of modes using the zero latency mode control instructions.

The prototype for this function is, as defined in `sysreg.h`:

```
void mode_change(const int _mode_spec);
```

The `_mode_spec` parameter is a bitmask of mode definitions defined in `sysreg.h`. These definitions are:

```
__MODE_ENA_AV_LATCH=0x1,  
__MODE_ENA_AR_SAT=0x2,  
__MODE_ENA_M_MODE =0x4,  
__MODE_ENA_TIMER=0x8,  
__MODE_DIS_AV_LATCH=0x100,  
__MODE_DIS_AR_SAT=0x200,  
__MODE_DIS_M_MODE=0x400,  
__MODE_DIS_INT =0x1000,
```

External Memory Read or Write

The `cc219x` compiler does not support multiple data pages and, therefore, cannot normally access the ADSP-219x DSP's external memory. The ADSP-219x DSP is a 16-bit chip and, therefore, integers and pointers are 16-bit wide and cannot hold the full range of data memory addresses. The alternative of making integers and pointers 32-bit wide is ruled out by lack of hardware support. The possibility of doing all integer and pointer updating by a subroutine is too inefficient for DSP work.

If you were to write a function to access external memory in assembly language, such a function would need to be passed the value to set the `DMPG` register to in order to access the required data. The same would be required in C because of the pointer size problems mentioned above.

The inclusion of `sysreg.h` allows the use of functions that will generate efficient inline instructions to implement read and write of values from and to external memory.

The prototypes of the functions are defined in `sysreg.h`:

```
int external_memory_read(int DMPG_val, int* addr);
void external_memory_write(int DMPG_val, int* addr, int val);
```

The `DMPG_val` parameter is the value of the upper 8 bits of the 24-bit external memory address. The `addr` parameter is a pointer to the external memory. The `val` parameter to `external_memory_write` is the value to be written to external memory. For example,

```
#include <sysreg.h>
section("external_memory_section")
static int GlobalTable[256];

int main() {
    int page, read_value, value_to_write = 0;
    asm("%0 = PAGE(GlobalTable);
    " : "=e"(page): : );
    external_memory_write(page, &GlobalTable[0], value_to_write);
    read_value = external_memory_read(page, &GlobalTable[1]);
    return read_value;
}
```

ETSI Support

The ETSI (European Telecommunications Standards Institute) support for ADSP-219x processors is a collection of functions that provides high performance implementations for operations commonly required by DSP applications. These operations provided by the ETSI library (`libetsi.dlb`) and compiler built-in functions (defined in `ETSI_fract_arith.h`) include support for fractional, or fixed point, arithmetic. The results obtained from use of these operations have well defined overflow and saturation conditions. The ETSI support operations are Analog Devices extensions to ANSI C standard.

The ETSI support contains functions that you can call from your source program. The following topics describe how to use this support.

- [“ETSI Support Overview” on page 1-92](#)
- [“Calling ETSI Library Functions” on page 1-94](#)
- [“Using the ETSI Built-In Functions” on page 1-95](#)
- [“Linking ETSI Library Functions” on page 1-95](#)
- [“Working with ETSI Library Source Code” on page 1-96](#)
- [“ETSI Support for Data Types” on page 1-96](#)
- [“ETSI Header File” on page 1-97](#)

ETSI Support Overview

The use of fractional arithmetic is vital for many applications on DSP processors as information can be held more compactly than in floating point. It would take 24 bits in floating-point format to match the precision of 16-bit fractional data. Also, control of normalization and precision is more complex with floating point. Many DSPs do not include hardware support for floating-point arithmetic and these operations are therefore very expensive in both code size and performance terms for such DSPs.

Fractional data has a representation similar to that of integers except that while an integer value is considered to have a decimal point to the right of the least significant bit, a fractional value is considered to have a decimal point to the left of the most significant bit. Fractional values are usually held in 16-bit or 32-bit “containers”. In each case, signed values are in the range $[-1.0, +1.0)$.

The bit operations on fractional data are identical to those on integer data, but there are three aspects of the result that are normally treated differently:

1. **MSB extraction.** Multiplication is a widening operation, thus multiplying a 16-bit value by another 16-bit value produces a 32-bit result. If a 16-bit integer result is required then this is taken to be the least significant 16 bits of the result, and the upper 16 bits are regarded as overflow. For a fractional operation the upper 16 bits would represent a 16-bit result, and the lower 16 bits would be regarded as an underflow.
2. **Duplicate sign bit elimination.** Following a multiplication of two 16-bit values the nature of the representation results in two “sign bits” in the result. For normal integer arithmetic this causes no problem, but for fractional arithmetic a shift left by one is required to normalize the result.
3. **Saturation.** If we perform an arithmetic operation that would cause us to overflow, it can be useful to return the maximum (appropriately signed) number that can be represented in the result register. The alternatives which include firing an interrupt, saying the result is undefined and is some other number, usually look less attractive to DSP programmers.

These fractional operations can often be done at no extra cost to normal integer operations on DSPs using special instructions or modes of operation.

The C programming language does not include a basic type for fractional data, and rather than introduce a non-standard type, Analog Devices defines `fract16` and `fract32` in terms of appropriately-sized integer data types and provides sets of basic intrinsic functions which perform the required operations. These look like library function calls, but are specially recognized by the compilers, which generate short sequences or

single instructions, exploiting any specialized features, which may be available on the architecture. An important aspect of this is that the compiler optimizer is not inhibited in any way by the use of these intrinsics.

Because of the varying nature of the architectures the basic intrinsic functions just discussed cannot be standardized across all the architectures. However, a set of standard functions for manipulating fractional data has been defined by the ITU (International Telecommunications Union) and ETSI (European Telecommunications Standards Institute).

Referred to as the ETSI Standard Functions, these have been very widely used to implement Telecommunications packages such as GSM, EFR and AMR Vocoders, and have become a de-facto industry standard. These functions have been implemented on ADSP-219x DSPs.

The ETSI standard is aimed at DSP processors with 16-bit inputs, saturated arithmetic and 32-bit accumulators.

Calling ETSI Library Functions

To use an ETSI 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. The names and arguments for each function appear in the section [“ETSI Header File” on page 1-97](#).

Like other functions you use, ETSI functions should be declared. Declarations are supplied in the header file `ETSI_fract_arith.h`, which must be included in any source files where ETSI functions are called. The function names are C function names. If you call C run-time library functions from an assembly language program, you must use the assembly version of the function name—prefix an underscore on the name. For more information on naming conventions, see the section [“C/C++ and Assembly Language Interface” on page 1-123](#).

The standard reference code for the ETSI functions use the `Carry` and `Overflow` flag global variables to keep track of any carry or overflow as a result of using on of the ETSI functions. With the ETSI functions provided by Analog Devices, this can be switched off by compiling with `__NO_ETSI_FLAGS` defined in the compiler command line.

In fact, this is the default for the ADSP-219x DSP implementation. If the user wishes to keep track of these flags, for debugging purposes, then they should compile with `__NO_ETSI_FLAGS` set to zero. This will mean that the user will be using the functions in accordance with the ETSI standard, but this will result in a reduced performance.

Using the ETSI Built-In Functions

Some of the ETSI functions have been implemented as part of cc219x compiler's set of built-in functions. For information on how to use these functions, refer to [“Compiler Built-in Functions” on page 1-85](#). These built-in implementations will be automatically defined when header file `ETSI_fract_arith.h` is included.

Linking ETSI Library Functions

When your C/C++ code calls an ETSI function that is not implemented using a compiler built-in, the call creates a reference that the linker resolves when linking. This requires the linker to be directed to link with the ETSI library, `libetsi.dlb`, in the `219x\lib` directory, which is a sub-directory of the VisualDSP++ installation directory. This is done automatically when using the default Linker Description File (LDF) for ADSP-219x processor targets, as these specify that `libetsi.dlb` will be on each link line.

If not using default `.LDF` files, then either add `libetsi.dlb` to the `.LDF` file which is being used, or alternatively use the compiler's `-letsi` switch to specify that `libetsi.dlb` is to be added to the link line.

Working with ETSI Library Source Code

The source code for functions and macros in the ETSI library is provided with your VisualDSP++ software. By default, the installation program copies the source code to a subdirectory of the directory where the run-time libraries are kept named `219x\lib\src\libetsi_src`. Each function is kept in a separate file. The file name is the name of the function with the extension `.asm`. If you do not intend to modify any of the functions, you can delete this directory and its contents to conserve disk space.

The source code is provided so you can customize specific functions for your own needs. To modify these files, you need proficiency in ADSP-219x assembly language and an understanding of the run-time environment, as explained in [“C/C++ and Assembly Language Interface” on page 1-123](#).

Before you make 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.

ETSI Support for Data Types

ETSI functions support `fract16` and `fract32` data types as follows:

- `fract16` is a 16-bit fractional data type (1.15 format) having a range of `[-1.0, +1.0)`. This is defined in the C/C++ language as

```
typedef short fract16
```

- `fract32` is a 32-bit fractional data type (1.31 format) having a range of `[-1.0, +1.0)`. This is defined in the C/C++ language as

```
typedef long fract32
```

ETSI Header File

The following table provides a summary of the functions provided by the ETSI library, as defined in the header file `ETSI_fract_arith.h`.

Table 1-11. ETSI Library Function

Description	Prototype
Short absolute	<code>fract16 abs_s (fract16)</code>
Short add	<code>fract16 add (fract16, fract16)</code>
Short division	<code>fract16 div_s (fract16, fract16)</code>
Long division	<code>fract16 div_l (fract32, fract16)</code>
Extract high (most significant 16 bits)	<code>fract16 extract_h (fract32)</code>
Extract low (least significant 16 bits)	<code>fract16 extract_l (fract32)</code>
Multiply and accumulate with rounding	<code>fract16 mac_r (fract32, fract16, fract16)</code>
Multiply and subtract with rounding	<code>fract16 msu_r (fract32, fract16, fract16)</code>
Short multiply	<code>fract16 mult (fract16, fract16)</code>
Multiply with rounding	<code>fract16 mult_r (fract16, fract16)</code>
Short negate	<code>fract16 negate (fract16)</code>
Long normalize	<code>fract16 norm_l (fract16)</code>
Short normalize	<code>fract16 norm_s (fract16)</code>
Round	<code>fract16 round (fract16)</code>
Saturate	<code>fract16 saturate (fract32)</code>
Short shift left	<code>fract16 shl (fract16, fract16)</code>
Short shift right	<code>fract16 shr (fract16, fract16)</code>
Shift right with rounding	<code>fract16 shr_r (fract16, fract16)</code>

C/C++ Compiler Language Extensions

Table 1-11. ETSI Library Function (Cont'd)

Description	Prototype
Short subtract	fract16 sub (fract16, fract16)
Long absolute	fract32 L_abs (fract32)
Long add	fract32 L_add (fract32, fract32)
Long add with carry	fract32 L_add_c (fract32, fract32)
16-bit variable -> most significant bits (least significant bits zeroed)	fract32 L_deposit_h (fract16)
16-bit variable -> least significant bits (sign extended)	fract32 L_deposit_l (fract16)
Multiply and accumulate	fract32 L_mac (fract32, fract16, fract16)
Multiply and accumulate without saturation	fract32 L_macNs (fract32, fract16, fract16)
Multiply both the most significant bits and the least significant bits of a long, by the same short	fract32 L_mls (fract32, fract16)
Multiply and subtract	fract32 L_msu (fract32, fract16, fract16)
Multiply and subtract without saturation	fract32 L_msuNs (fract32, fract16, fract16)
Long multiply	fract32 L_mult (fract16, fract16)
Long negate	fract32 L_negate (fract32)
Long saturation	fract32 L_sat (fract32)
Long shift left	fract32 L_shl (fract32, fract16)
Long shift right	fract32 L_shr (fract32, fract16)
Long shift right with rounding	fract32 L_shr_r (fract32, fract16)
Long subtract	fract32 L_sub (fract32, fract32)
Long subtract with carry	fract32 L_sub_c (fract32, fract32)
Compose long	fract32 L_Comp (fract16, fract16)

Table 1-11. ETSI Library Function (Cont'd)

Description	Prototype
Multiply two longs	fract32 Mpy_32 (fract16, fract16, fract16, fract16)
Multiply short by a long	fract32 Mpy_32_16 (fract16, fract16, fract16)
Extract a long from two shorts	void L_Extract (fract32, fract16*, fract16*)
Fract integer division of two longs	fract32 Div_32 (fract32, fract16, fract16)

Pragmas

The C/C++ compiler for ADSP-219x DSPs supports a number of pragmas. Pragmas are implementation-specific directives that modify the compiler's behavior. The C/C++ compiler supports pragmas for:

- Arranging alignment of data
- Defining functions that can act as interrupt handlers
- Changing the optimization level, midway through a module
- Changing how an externally visible function is linked

The following sections describe the pragmas that support these features.

- [“Data Alignment Pragma - ALIGN NUM” on page 1-100](#)
- [“Interrupt Handler Pragma” on page 1-100](#)
- [“Altregisters Pragma” on page 1-101](#)
- [“Optimization Level Pragmas” on page 1-101](#)
- [“Linkage Pragmas” on page 1-101](#)

Data Alignment Pragma - ALIGN NUM

The `align num` pragma may be used before variable and field declarations. It applies to the variable or field declaration that immediately follows the pragma. Use of this pragma causes the compiler to generate the next variable or field declaration aligned on a boundary specified by `num`.

The `align num` pragma is useful for declaring arrays that need to be on a circular boundary. Such arrays might be required to make use of a bitreversal sorting algorithm that is implemented using the ADSP-219x processor's DAG1 bit reversal mode.

```
#pragma align 256
int arr[128];
```

Interrupt Handler Pragma

The `interrupt` pragma may be used before a function declaration or definition. It applies to the function declaration or definition that immediately follows the pragma. Use of this pragma causes the compiler to generate the function code so that it may be used as a self dispatching interrupt handler.

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 a return from interrupt (RTI) instruction.

```
#pragma interrupt
void field_SIG()
{
    /* ISR code */
}
```

Altregisters Pragma

The `altregisters` pragma may be used in conjunction to the `interrupt-pragma` to indicate that the compiler can optimize the saving and restoring of registers through use of the secondary register sets. Note the use of the `altregisters` pragma is not safe when nested interrupts are enabled.

```
#pragma interrupt
#pragma altregisters
void field_SIG()
{
  /* ISR code */
}
```

Optimization Level Pragmas

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.

```
#pragma optimize_off
void non_op() { /* non-optimized code */ }

#pragma optimize_for_space
void op_for_sp() { /* code optimized for size */ }

#pragma optimize_for_speed
void op_for_sp() { /* code optimized for speed */ }
```

Linkage Pragmas

Linkage pragmas (`linkage_name` and `weak_entry`) change how a given global function or variable is viewed during the linking stage.

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.

```
_Pragma("linkage_name __realfuncname")  
void funcname ();
```

weak_entry Pragma

The `weak_entry` pragma may be used before a function or static variable declaration or definition. It applies to the function or variable declaration or definition that immediately follows the pragma. Use of this pragma causes the compiler to generate the function or variable definition with weak linkage.

```
#pragma weak_entry  
int w_var = 0;  
  
#pragma weak_entry  
void w_func(){}  

```

Preprocessor Features

The cc219x 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) comments
#warning directive
```

For more information about these extensions refer to [“Preprocessor Generated Warnings”](#) and [“Preprocessor Generated Warnings”](#) on page 1-85 .

Predefined Preprocessor Macros

The cc219x compiler defines a number of macros to produce 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.

The predefined preprocessor macros are:

<code>__ADSP21XX__</code>	cc219x always defines <code>__ADSP21XX__</code> and <code>__ADSP219X__</code> as 1.
<code>__ADSP2191__</code>	cc219x defines <code>__ADSP2191__</code> as 1 when you compile with the <code>-2191</code> command-line switch.
<code>__ADSP2192_12__</code>	cc219x defines <code>__ADSP2192_12__</code> as 1 when you compile with the <code>-2192-12</code> command-line switch.
<code>__ANALOG_EXTENSIONS__</code>	cc219x defines <code>__ANALOG_EXTENSIONS__</code> as 1, unless you compile with <code>-pedantic</code> or <code>-pedantic-errors</code> .
<code>__cplusplus</code>	cc219x defines <code>__cplusplus</code> as 1 when you compile in C++ mode.

Preprocessor Features

<code>__DATE__</code>	The preprocessor expands this macro into the current date as a string constant. The date string constant takes the form <code>mm dd yyyy</code> (ANSI standard).
<code>__DOUBLES_ARE_FLOATS__</code>	<code>cc219x</code> always defines <code>__DOUBLES_ARE_FLOATS__</code> as 1.
<code>__ECC__</code>	<code>cc219x</code> always defines <code>__ECC__</code> as 1.
<code>__EDG__</code>	<code>cc219x</code> always defines <code>__EDG__</code> as 1. This signifies that an Edison Design Group front end is being used
<code>__EDG_VERSION__</code>	<code>cc219x</code> always defines <code>__EDG_VERSION__</code> as an integral value representing the version of the compiler's front end.
<code>__FILE__</code>	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 <code>cc219x</code> command line or in a preprocessor <code>#include</code> command (ANSI standard).
<code>_LANGUAGE_C</code>	<code>cc219x</code> always defines <code>_LANGUAGE_C</code> as 1 when compiling C or C++ source.
<code>__LINE__</code>	The preprocessor expands the <code>__LINE__</code> macro into the current input line number as a decimal integer constant (ANSI standard).
<code>__NO_BUILTIN</code>	<code>cc219x</code> defines <code>__NO_BUILTIN</code> as 1 when you compile with the <code>-no-builtin</code> command-line switch.
<code>_NO_LONG_LONG</code>	<code>cc219x</code> always defines <code>_NO_LONG_LONG</code> as 1 for C and C++ source. This definition signifies no support is present for the <code>long long int</code> type.
<code>__SIGNED_CHARS__</code>	<code>cc219x</code> defines <code>__SIGNED_CHARS__</code> as 1 unless you compile with the <code>-unsigned-char</code> command-line switch.
<code>__STDC__</code>	<code>cc219x</code> defines <code>__STDC__</code> as 1, unless you compile with <code>-traditional</code> (ANSI standard).
<code>__STDC_VERSION__</code>	<code>cc219x</code> defines <code>__STDC_VERSION__</code> as 199409L, unless you compile with <code>-traditional</code> (ANSI standard).
<code>__TIME__</code>	The preprocessor expands this macro into the current time as a string constant. The time string constant takes the form <code>hh:mm:ss</code> (ANSI standard).

Header Files

A header file contains C or C++ declarations and macro definitions. Use the `#include` preprocessor directive to access header files for your program. Header file names have a `.h` extension. There are two main categories of header files:

- System header files declare the interfaces to the libraries supplied with the ADSP-219x product. Include them in your program for the definitions and declarations you need to access. Use angle brackets to indicate a system header file: `#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: `#include "file"`

Writing Macros

A macro is a name standing for a block of text that the preprocessor substitutes. Use the `#define` preprocessor command to create a macro definition. When the 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, it can be useful to define a macro that expands into a compound statement. You can define such a macro so it can be invoked in 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); \
```

Preprocessor Features

```
while ((p) != lim)          { \
    if (*(p)++ != ' ')      { \
        (p)--; \
        break; \
    } \
} \
}

/* SKIP_SPACES, enclosed macro */
#define SKIP_SPACES (p, limit) \
do { \
    char *lim = (limit); \
    while ((p) != lim) { \
        if (*(p)++ != ' ') { \
            (p)--; \
            break; \
        } \
    } \
} \
} while (0)
```

Enclosing the first definition within the `do {...} while (0)` pair changes the macro from expanding to a compound statement to expanding to a single statement. With the macro expanding to a compound statement, you would sometimes need to omit the semicolon after the macro call in order to have a legal program. This leads to a need to remember whether a function or macro is being invoked for each call and whether the macro needs a trailing semicolon or not.

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

```
/* SKIP_SPACES, enclosed macro, ends without ';' */
if (*p != 0)
    SKIP_SPACES (p, lim);
else ...
```

This expands to:

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

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

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

The above code is not legal C or C++ syntax.

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](#).

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.

Preprocessor Features

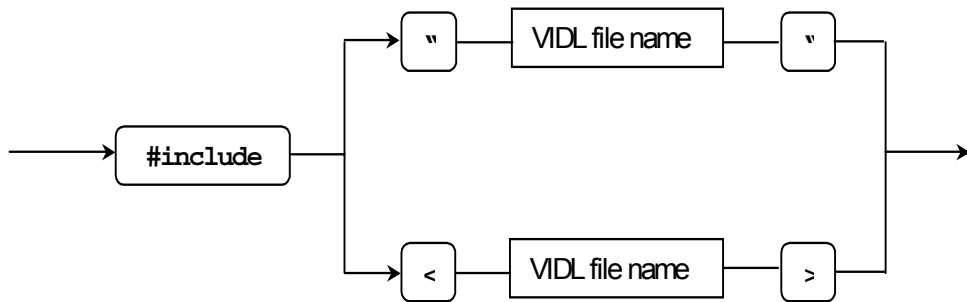


Figure 1-2. #INCLUDE Syntax Diagram

For more information, refer to the *VisualDSP++ Component Software Engineering User's Guide*.

C/C++ Run-Time Model and Environment

This section provides a full description of the ADSP-219x 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 ADSP-219x DSPs. Assembly routines linked to C/C++ routines must follow these conventions.



ADI recommends that assembly programmers maintain stack conventions.

Using the Run-Time Header

The run-time header is an assembly language procedure that initializes the processor and sets up processor features to support the C run-time environment. The default run-time header source code for the ADSP-219x DSPs is in the `219x_hdr.asm` file. This run-time header performs the following operations:

- Initializes the C run-time environment
- Calls your `main` routine
- Calls `exit` routine, defined in the C run-time library (`libc.dlb`), if `main` returns.
- Defines system halt instruction called from `exit` routine.

Interrupt Table and Interface

The interrupt table is an assembly language set of functions defined in named sections. These sections get placed appropriately in the Linker Description File (LDF) to be executed at interrupt vector addresses. The default code for the ADSP-219x DSP's interrupt table is defined in `219x_int_tab.asm`.

The default interrupt table uses the following external symbols:

<code>_lib_int_table</code>	Static table holding interrupt information defined in the C run-time library
<code>__lib_int_determiner</code>	An interrupt dispatcher defined in the C run-time library
<code>____system_start</code>	C run-time initialization defined in the run-time header

The `219x_int_tab` file contains a section of code for each hardware interrupt. The `.LDF` file places these code sections in the correct interrupt vector slots for each interrupt.

If an interrupt occurs, program execution begins at the interrupt vector addresses. Program execution causes a jump to `__lib_int_determiner` in the default vector code. If `__lib_int_determiner` finds (by inspecting `__lib_int_table`) a handler set for the interrupt, it will call the handler. `__lib_int_determiner` saves and restores all scratch registers around the handler call. The function `__lib_int_determiner` terminates by executing a return from interrupt (RTI) instruction, which restores program execution to the point at which the interrupt was raised.

A handler for an interrupt or signal is set using the `interrupt` or `signal` C run-time library functions. These functions pass the signal name and a handler function pointer as parameters. The signal macro names are defined in `signal.h`.

The default interrupt vector code may be replaced with custom code by modifying or creating a new piece of code to be placed at the vector addresses. This is usually done by copying the default `219x_int_tab.asm` file and `.LDF` file into your project and modifying them as required.

An `interrupt` pragma defined function can be placed in the interrupt vector code directly or be jumped to from the vector if it does not fit in the interrupt vector space (see [“Interrupt Handler Pragma” on page 1-100](#)).



The reset vector code, which is placed at address zero (0) and does a jump to `____system_start`, should not be replaced.

Stack Frame

The stack frame (or activation record) provides for the following activities:

- space for local variables for the current procedure. For the compiler, this includes temporary storage as well as that required for explicitly declared user automatic variables.
- place to save linkage information such as return addresses, location information for the previous caller's stack frame and to allow this procedure to return to its caller
- space to save information that must be preserved and restored
- arguments passed to the current procedure

In addition, if this is not a leaf procedure (if it is going to call other procedures), its stack frame also contains outgoing linkage and parameter space:

- space for the arguments to the called procedure.
- space for the callee to save basic linkage information.

Figure 1-3 provides a general overview of the stack. Note that the stack grows downward on the page. Because the stacks grow towards smaller addresses, higher addresses are found in the upwards direction. “[Stack Frame Description](#)” on page 1-113

Incoming Arguments	
Linkage Information	← FP
Linkage Information and Temporaries	
Save Area (for caller info)	
Outgoing Arguments	
Free Space	← SP

Figure 1-3. ADSP-219x DSP Stack

The stack resides in primary Data Memory (DM). It is controlled by a pair of pointers: a Stack Pointer (SP), which identifies the boundary of the in-use portion of the stack space, and a Frame Pointer (FP), which provides stable addressing to the current frame.

For increased efficiency the ADSP-219x DSP run-time model assumes the 32 words past the stack pointer are available for use within a leaf function (functions that make no calls) and are *protected* from changes by an interrupt handler. This provides easy access to temporary space for use within simple leaf functions without the overhead of establishing a stack frame.

Stack Frame Description

This section describes the stack as shown in [Figure 1-3 on page 1-112](#).

Incoming Arguments

The memory area for incoming arguments begins at the `FP` value +1. Argument words are mapped by ascending addresses, so the second argument word is mapped at `FP+2`.

Linkage Information

The return address is saved on the hardware stack by the `CALL` instruction. In the called function, the address can then be popped from the hardware stack and saved as part of the stack frame. This information is used by the debugger to generate call stack debug information for source level debugging. Saving the return address on the stack frame is also useful in avoiding overflowing the finite hardware call stack, when using recursion for example. The value stored on the stack gets pushed back on the hardware call stack before the return of the function. The compiler detects recursive routines and also offers a switch to avoid overflowing the hardware call stack, called `-no_hardware_pc_stack`, which might be required for highly nested software.

Local Variables and Temporaries

Space for a register save area and local variables/temporaries is allocated on the stack by the function prologue. Local variables and temporaries are typically placed first in this area so they can be addressed with the smallest offsets from `FP`. The register save area is located at the farthest end of this area and can be accessed by `SP`-relative addressing.

Outgoing Arguments

Outgoing arguments are located at the top of the stack prior to the call. Space may be pre-allocated or claimed at the time of each call.

Free Space

Space below `SP` is normally considered free and unprotected; it is available for use (in growing the stack) at any time, synchronously or asynchronously (the latter for interrupt handling). However, on the ADSP-219x DSPs, the 32 words past the `SP` are reserved as a protected temporary space for use within a procedure.

General System-Wide Specifications

Some general specifications that apply to the stacks are:

- The stacks grow down in memory from higher to lower addresses.
- The current frames' "base" is addressed by the `FP` register.
- The first free word in each stack is addressed by the `SP` register.

 Data can be pushed onto the stack by executing an instruction like the following one for the ADSP-219x DSPs:

```
DM (I4 += M5) = rej.
```

- The return address of the caller is stored at offsets -1 and -2 from the address carried by the current `FP`, if it is stored on the stack.
- The linkage back to the previous stack frame is stored at offset 0 from the current `FP`.

At a procedure call, the following must be true:

- For the ADSP-219x DSPs, no space beyond the `SP` must be in use.
- There must be at least one free slot on the `PC` stack to hold the return address.

At an interrupt, the following must be true:

- Space beyond the `SP` must be available.
- For the ADSP-219x DSPs, the first 32 words beyond the `SP` must be protected; the interrupt routine should decrement the `SP` by 32 and then restore the original value of `SP` prior to return.

Return Values

Return values always use registers. Single word return values come back in register `AX1`. Double word return values are stored in `SR1:0`, with the most significant part in `SR1`.

If the return value is larger than two words, then the caller must allocate space and pass the address in, as a “hidden argument”. The register `I0` is used for this purpose.

Procedure Call and Return

To call a procedure:

1. Evaluate the arguments and push them onto the stack.
2. Call the procedure.
3. On return, remove arguments if desired.

On Entry:

1. Save the old FP and set FP to current SP.
2. If `-no_hardware_pc_stack` is specified, debugging, pop the PC stack and store it on the main stack.
3. Move the SP to create space for the new frame.
4. If `-g` is specified but `-no_hardware_pc_stack` is not specified, push the return address back onto the hardware stack.

After this step, the new frame is officially in place.

5. Continue saving registers, and then executing the procedure.

A leaf procedure which does not require much stack space might choose to omit steps (1) and (2), operating without its own stack frame. On the ADSP-219x DSPs, the 32 words of protected space beyond the SP can be used for temporary storage.

To Return from a Procedure:

1. If the hardware PC stack is not used, the return address must be loaded from the stack and restored.
2. Restore miscellaneous saved registers.
3. Place the return value in the correct register (if not there already).
4. Restore FP for the previous frame.
5. Reset SP to remove the frame for procedure.
6. Return to the caller.

Miscellaneous Information

This section contains a number of miscellaneous aspects of the design that may be helpful in understanding stack functionality.

- Procedures without prototypes can be called successfully, provided the argument types correspond properly. Prototypes are always good programming practice. Programs that call library subroutines should always include header files.
- There is no special interface for calling system library functions. They use the standard calling convention.

Register Classification

This section describes all of the ADSP-219x processor registers. Registers are listed in order of preferred allocation by the compiler

Callee Preserved Registers (“Preserved”)

Registers I2, I3, I7, M0, M2 and M4 are preserved. A subroutine which uses any of these registers must save (preserve) it and restore it.

Dedicated Registers

Certain registers have dedicated purposes and will not be used for other things. Compiled code and libraries expect the dedicated registers to be correct.

Caller Save Registers (“Scratch”)

All registers not preserved or dedicated are *scratch*. A subroutine may use a scratch register without having to save it.

Circular Buffer Length Registers

Registers `L0` through `L7` are the circular buffer length registers. The compiler assumes that these registers contain zero, which disables circular buffering. They **must** be set to zero when compiled code is executing to avoid incorrect behavior. There is no restriction on the value of an `L` register when the corresponding `I` register has been reserved from compiler use. See “[-reserve register\[, register ...\]](#)” on [page 1-39](#) for information on reserving registers.

Mode Status (MSTAT) Register

The C runtime initializes the `MSTAT` register as part of the run-time header code. The compiler and run-time libraries assume to be running in these preset modes. If you change any of the modes listed in [Table 1-12 on page 1-118](#), ensure that they are reverted before calling C/C++ compiled functions or functions from the C run-time library. Failure to revert to the default modes may cause applications to fail when running.

Table 1-12. MSTAT Register Modes

Mode	Description	State
<code>SEC_REG</code>	Secondary Data Registers	disabled
<code>BIT_REV</code>	Bit-reversed address output	disabled
<code>AR_SAT</code>	ALU saturation mode	disabled
<code>M_MODE</code>	MAC result mode	Integer Mode, 16.0 format
<code>SEC_DAG</code>	Secondary DAG registers	disabled

Complete List of Registers

The following tables describe all of the registers for the ADSP-219x DSPs.

- [Table 1-13](#) lists the data register's file registers
- [Table 1-14](#) lists the DAG1 registers
- [Table 1-15](#) lists the DAG2 registers
- [Table 1-16](#) lists miscellaneous registers



You must **always** specify **all parts** of the SR and MR register groups. For example, either MR1 or SR0 is valid, but either SR or MR by itself is **not** valid.

Table 1-13. List of Data Register File Registers

Register	Descriptuon	Notes
AX0	scratch	
AX1	scratch; single-word return	
AY0	scratch	
AY1	scratch	
AR	scratch;	
AF	scratch	
MX0	scratch	
MX1	scratch	
MY0	scratch	
MY1	scratch	
MR1:0	scratch	
MR2	scratch	

Table 1-13. List of Data Register File Registers (Cont'd)

Register	Descriptuon	Notes
PX	scratch	
SB	scratch	
SE	scratch	
S1	scratch	
SR1:0	scratch; double-word return	
SR2	scratch	

Table 1-14. List of DAG1 Registers

Register	Descriptuon
I0	scratch
I1	scratch
I2	preserved
I3	preserved
M0	preserved
M1	scratch
M2	preserved
M3	scratch
L0-3	not used, must be zero
B0-3	not used
DMPG1	preserved

Table 1-15. List of DAG2 Registers

Register	Descriptuon
I4	Dedicated: SP
I5	Dedicated: FP
I6	scratch
I7	preserved
M4	preserved
M5	dedicated: -1
M6	scratch
M7	scratch
L4-7	not used, must be zero
B4-7	not used
DMPG2	dedicated, must not change

Table 1-16. Miscellaneous Registers

Register	Descriptuon
ASTAT	scratch
CCODE	preserved; (0x8 ALU result sign default in C/C++ code)
CNTR	scratch
ICNTL	scratch
IJPG	scratch
IMASK	scratch
IRPTL	scratch
LPSTACKA	scratch

Table 1-16. Miscellaneous Registers (Cont'd)

LPSTACKP	scratch
MSTAT	scratch
STACKA	scratch
STACKP	scratch
STACKA	scratch

C/C++ and Assembly Language Interface

This section describes how to call assembly language subroutines from within C or C++ programs and C or C++ functions from within assembly language programs.



Before attempting to perform either of these calls, be sure to familiarize yourself with the information about the C/C++ run-time model (including details about the stack, data types, and how arguments are handled) contained in [“C/C++ Run-Time Model and Environment”](#) on page 1-109.

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 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.

C/C++ and Assembly Language Interface

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).

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. If you're using the function from assembly programs as well, you might want your function's name to be just as you write it. Then you'll also need to tell the C/C++ compiler that it's an `asm` function, by placing `'extern "asm" {}'` around the prototype.

The C/C++ runtime determines that all function parameters are passed on the stack. A good way to observe and understand how arguments are passed is to write a dummy function in C or C++ and compile it using the `-save-temps` command-line switch (see [on page 1-40](#)). The resulting compiler generated assembly file (`.s`) can then be viewed.

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, int d, int e) {
    //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:

```
// PROCEDURE: _asmfunc
.global _asmfunc;
_asmfunc:
    SI = DM(I4 + 4);
    I0 = SI ;
    AX1 = DM(I4 + 2);
    SI = DM(I4 + 1);
    AX0 = DM(I4 + 3);
    DM(_global_b) = AX1;
    DM(_global_a) = SI;
    DM(_global_b+1) = AX0;
    RTS (DB);
    AX1 = 12345;
    DM(_global_p) = I0;
_asmfunc.end
```



For a more complicated function, you might find it useful to follow the general run-time model, and use the run-time stack for local storage, etc. A simple C program, passed through the compiler, will provide a good template to build on. Alternatively, you may find it just as convenient to use local static storage for temporaries.

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 1-123](#), 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 names or variable names) in assembly routines and use assembly symbols in C routines. This section describes how to name C/C++ and assembly symbols and shows how to use C/C++ and assembly symbols.

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 your assembly routine, declare it as global in the C/C++ program and import the symbol into the assembly routine by declaring the symbol with the `.EXTERN` assembler directive.

The C++ language performs name mangling on function names it defines according to the output and input parameter types of the function. If calling into a C++ defined function from assembly code, the `.EXTERN` symbol will need to be the mangled C++ output name. This is best retrieved by looking at the compiler's assembly output for the C++ source that defines the required function.

C/C++ and Assembly Language Interface

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/C++ program.

Alternatively, the cc219x compiler provides an “asm” linkage specifier (used similarly to the “C” linkage specifier of C++), which when used, removes the need to add an underscore prefix to the symbol that is defined in assembly.

Table 1-17 shows several examples of the C/C++ and assembly interface naming conventions.

Table 1-17. C/C++ Naming Conventions for Symbols

In C/C++ Program	In Assembly Subroutine
<code>int c_var; /*declared global*/</code>	<code>.extern _c_var;</code>
<code>void c_func(); /* in C code */</code>	<code>.extern _c_func;</code>
<code>void cpp_func(void); /* in C++ source */</code>	<code>.extern _cpp_func__Fv;</code>
<code>extern int asm_var;</code>	<code>.global _asm_var;</code>
<code>extern void asm_func();</code>	<code>.global _asm_func; _asm_func:</code>
<code>extern "asm" void asm_func();</code>	<code>.global asm_func; _asm_func:</code>

C++ Programming Examples

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

- [“Using Fract Type Support” on page 1-129](#)
- [“Using Complex Number Support” on page 1-130](#)

By default, the `cc219x` 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 the VisualDSP++ environment.

For example, the following command line

```
cc219x -c++ source.cpp -2191
```

runs `cc219x` with:

`-c++`

Specifies that 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.

`-2191`

Specifies that the compiler should produce code suitable for the ADSP-2191 DSP.

Using Fract Type Support

[Listing 1-1 on page 1-130](#) 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.

C/C++ and Assembly Language Interface

For more information about the fractional data type and arithmetic, see [“Fractional Type Support” on page 1-82](#).

Listing 1-1. Example Code: Using Fract Data Type — C++ code

```
#include <fract>
#define N 20
fract x[N] = {.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 Number 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;
    real = zreal * zreal;
    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 Libraries 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 `cc219x` compiler.
- [“Documented Library Functions” on page 2-21](#)
It tabulates the functions that are defined by ANSI standard header files.
- [“C Run-Time Library Reference” on page 2-24](#)
It provides reference information about the C run-time library functions included with this release of the `cc219x` compiler.

The `cc219x` compiler provides a broad collection of library functions including those required by the ANSI standard and additional Analog-supplied functions of value for DSP programming. 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 library. Chapter 3, [“DSP Run-Time Library”](#) describes a number of signal processing, 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 Libraries Guide

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

This section provides information on the following topics:

- [“Calling Library Functions” on page 2-3](#)
- [“Using the Compiler’s Built-In C Library Functions” on page 2-4](#)
- [“Linking Library Functions” on page 2-5](#)
- [“Working With Library Source Code” on page 2-7](#)
- [“Working With Library Header Files” on page 2-8](#)
- [“Abridged C++ Library Support” on page 2-14](#)

For information on the C library’s contents, see [“Documented Library Functions” on page 2-21](#). For information on the Abridged C++ library’s contents, see [“Abridged C++ Library Support” on page 2-14](#) and on-line Help.

Calling Library Functions

To use a C/C++ library function, call the function by name and give the appropriate arguments. The name and arguments for each function appear on the function’s reference page. The reference pages appear in the [“Documented Library Functions” on page 2-21](#) and in the **C++ Run-Time Library** topic of the on-line Help.

Like other functions you use, library functions should be declared. Declarations are supplied in header files. For more information about the header files see [“Working With Library Source Code” on page 2-7](#).

The 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 on the name). For more information on the naming conventions, see [“C/C++ and Assembly Language Interface” on page 1-123](#).



You can use `elfar` (the archiver), described in the *VisualDSP++ 3.0 Linker and Utilities Manual for ADSP-218x and ADSP-219x DSPs*, to build library archive files of your own functions.

Using the Compiler's Built-In C Library 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, in-line assembly code is faster than an library routine, and it does not incur the calling overhead.

To use built-in functions, your source must include the required standard include file. For the abs functions this would require `stdlib.h` to be included. There are built-in functions used to define some ANSI C `math.h`, `string.h` and `stdlib.h` functions. There are also built-in functions to support various ANALOG extensions to the ANSI standard defined in the include file `math_builtins.h`. Not all built-in functions have a library alternate definition. Therefore, the failure to use the required include files can result in your program build failing to link.

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

Linking Library Functions

The C/C++ run-time library is organized as several libraries which are catalogued in [Table 2-1 on page 2-6](#). The libraries and start-up files are installed within the subdirectory `...\219x\lib` of your VisualDSP++ installation.

Several library files are built twice: once for single-threaded environments, and once for multi-threaded environments. The multi-threaded versions have a "mt" suffix. Another variant for some of the libraries is also provided that avoids a hardware anomaly involving instruction type 32a; these libraries have a "_type32aworkaround" 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-21<your_target>.ldf`).



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

For example, the switches `-lc -lets` will add the C library `libc.dlb` and the ETSI support library `libetsi.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 ADSP-218x and ADSP-219x DSPs*.

[Table 2-1](#) briefly describes the ADSP-219x DSP library functions.

C and C++ Run-Time Libraries Guide

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

219x\lib Directory	Description
2192-12_int_tab.doj	Default interrupt vector code for ADSP-2192-12 DSP
219x_int_tab.doj	Default interrupt vector code for ADSP-219x DSPs
219x_hdr.doj	Start-up file — set-up routines and call <code>main()</code>
219x_cpp_hdr.doj 219x_cpp_mt_hdr.doj	C++ start-up file — set-up routines and call <code>main()</code>
219x_exit.doj	Dummy exit object for backwards compatibility.
219x_ezkit_hdr.doj	Start-up file — set-up routines, call <code>main()</code> , and enable use of the EZ-kit monitor program
219x_cpp_ezkit_hdr.doj 219x_cpp_mt_ezkit_hdr.doj	C++ start-up file — set-up routines, call <code>main()</code> , and enable use of the EZ-kit monitor program
libc.dlb libcmt.dlb libc_type32aworkaround.dlb libcmt_type32aworkaround.dlb	C run-time library
libcpp.dlb libcppmt.dlb libcpp_type32aworkaround.dlb libcppmt_type32aworkaround.dlb	C++ run-time library
libcpprt.dlb libcpprmt.dlb libcpprt_type32aworkaround.dlb libcpprmt_type32aworkaround.dlb	C++ run-time support library
libdsp.dlb libdsp_type32aworkaround.dlb	DSP library
libetsi.dlb	ETSI run-time library

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

219x\lib Directory	Description
libio.dlb libiomt.dlb libio_type32aworkaround.dlb libiomt_type32aworkaround.dlb	I/O library
libsim.dlb	Simulator library support



Ensure that the C++ library of run-time routines, `libcprprt.dlb`, is the last library specified on the link line.



If all the objects supplied to the driver have been built as C, but are referencing a C++ object which is in a library, the standard C++ libraries are not searched and the linker may issue an error concerning unresolved symbol(s). This can be avoided by using the compiler `flags-link` switch (see [on page 1-27](#)), which ensures that that the C++ libraries are linked from the default `.LDF` files:

```
flags-link -MD__cplusplus=1
```

Note that this problem will only occur if the C++ object is in a library. If it is in an object file, the compiler will recognize it as a C++ object and link with the C++ libraries.

Working With Library Source Code

The source code for the functions in the C run-time library is provided with your VisualDSP++ software. By default, the installation program copies the source code to a subdirectory of the directory where the run-time libraries are kept, named `...\219x\lib\src`. Each function is kept in a separate file. The file name is the name of the function with the appropriate extension for C or assembler source. If you do not intend to modify any of the run-time library functions and are not interested in using the source file reference, you can delete this directory and its contents to conserve disk space.

The source code is provided so you can customize any particular function for your own needs. To modify these files, you need proficiency in ADSP-219x DSP assembly language and an understanding of the run-time environment, as explained in [“C/C++ Run-Time Model and Environment” on page 1-109](#). Before you make any modifications to the source code, copy the source code to a file with a different filename and rename the function itself. Test the function before you use it in your system to verify that it is functionally correct.



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

Working With Library Header Files

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

A list of the header files that are supplied with this release of the `cc219x` compiler appears in [Table 2-1 on page 2-6](#). You should use a C standard text to augment the information supplied in this chapter.

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

Header	Purpose	Standard
<code>assert.h</code>	Diagnostics	ANSI
<code>ctype.h</code>	Character Handling	ANSI
<code>def2191.h</code>	Memory Map Register and System Definitions for ADSP-2191 DSPs	C Extension
<code>def2192-12.h</code>	Memory Map Register and System Definitions for ADSP-2192-12 DSPs	C Extension

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

Header	Purpose	Standard
def219x.h	Memory Map Definitions	C Extension
errno.h	Error Handling	ANSI
float.h	Floating Point	ANSI
limits.h	Limits	ANSI
locale.h	Localization	ANSI
math.h	Mathematics	ANSI
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
sysreg.h	Efficient system access	C Extension

C Run-Time Library Header File Descriptions

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

assert

The `assert.h` header file contains the `assert` macro.

ctype

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

def2191 – Memory Map Definitions

The `def2191.h` header file contains macro definitions for the ADSP-2191 processor's system addresses, and system register bits. These symbolic names can be used in programs to access specific system registers and system register bits in the ADSP-2191 DSP.

def2192-12 – Memory Map Definitions

The `def2192-12.h` header file contains macro definitions for the ADSP-2191-12 processor's system addresses, and system register bits. These symbolic names can be used in programs to access specific system registers and system register bits in the ADSP-2191-12 DSP.

def219x – Memory Map Definitions

The `def219x.h` header file contains macro definitions for a ADSP-219x processor's system addresses, and system register bits. These symbolic names can be used in programs to access specific system registers and system register bits in ADSP-219x DSPs.

errno

The `errno.h` header file provides access to `errno`. This facility is not, in general, supported by the rest of the library.

float

The format of floating point data types are defined in the header file `float.h`. The `FLT_ROUNDS` macro is defined as 4 in the `float.h` header file. The 4 indicates an implementation-defined rounding mode.

The C run-time environment defines the rounding mode for `float` variables as round-to-nearest.

limits

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

locale

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

math

This category 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 ADSP-219x processors, 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`. `HUGE_VAL` evaluates to the maximum positive value that the type `double` can support. The macros `EDOM` and `ERANGE`, defined in `errno.h`, are used by `math.h` functions to indicate domain and range errors.

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`.

C and C++ Run-Time Libraries Guide

setjmp

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

signal

The `signal.h` header file provides function prototypes for the standard ANSI `signal.h` routines and also for several ADSP-219x DSP's extensions such as `interrupt()` and `clear_interrupt()`.

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

stdarg

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

stddef

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

stdio

The implementation of the `stdio.h` routines provides for a simple interface with a host environment. The simulator provides internal host support for these routines. Calls to `stdio.h` routines are trapped by the simulator, and a host implementation provides the correct functionality. Applications that use the `stdio.h` functions should link with the library `libio.dlb` in the same way as linking with the C run-time library `libc.dlb`.

The I/O library is thread-safe but it is not interrupt-safe; the library should not therefore be called either directly or indirectly from an interrupt service routine.



The following facilities are not available in this release:

- The stream positioning functions `fgetpos`, `fseek`, `fsetpos`, `ftell`, and `rewind`
- The file handling functions `remove`, `rename`, `tmpfile`, and `tmpnam`
- The format specifier `%p`

stdlib

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

string

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

sysreg

The `sysreg.h` header file defines a set of functions that provide efficient system access to registers, modes and addresses not normally accessible from C source. See [“Compiler Built-in Functions” on page 1-85](#) for more information on these functions.

Abridged C++ Library Support

When in C++ mode, the cc219x compiler can call a large number of 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 2-14](#)
- [“C++ Header Files for Library Facilities” on page 2-17](#)
- [“Embedded Standard Template Library Header Files” on page 2-18](#)

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

Embedded C++ Library Header Files

The following sections provide a brief description of the header files in the Embedded C++ Library

complex

The `complex.h` header defines the template class `complex` and a set of associated arithmetic operators.

exception

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

fract

The `fract.h` header defines the `fract` data type, which supports fractional arithmetic, assignment, and type-conversion operations. The header file is fully described under [“C++ Header Files for Library Facilities” on page 2-17](#), and an example that demonstrates its use appears under [“C++ Programming Examples” on page 1-129](#).

fstream

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

iomanip

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

ios

The `ios.h` header defines several classes and functions for basic `iostream` manipulations.



Most of the `iostream` header files include `ios.h`.

iosfwd

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

iostream

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

C and C++ Run-Time Libraries Guide

istream

The `istream.h` header defines the `istream` class for `istream` extractions.



Most of the `istream` header files include `istream.h`.

new

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

ostream

The `ostream.h` header defines the `ostream` class for `ostream` insertions.

sstream

The `sstream.h` header defines the `stringstream`, `istringstream`, and `ostringstream` classes for various string object manipulations.

stdexcept

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

streambuf

The `streambuf.h` header defines the `streambuf` classes for basic operations of the `istream` classes.



Most of the `istream` header files include `streambuf.h`.

string

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



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

strstream

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

C++ Header Files for 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`.

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, and the effect of including one of the C++ header files listed in [Table 2-3](#) is the same as including the equivalent C standard library header file.

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

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.

C and C++ Run-Time Libraries Guide

Table 2-3. C++ Header Files for C Library Facilities (Cont'd)

Header	Description
<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 variable 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, but they are supported by the `cc219x` compiler in C++ mode.

The Embedded Standard Template Library headers are:

algorithm

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

deque

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

functional

The `functional.h` header defines numerous function objects.

hash_map

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

hash_set

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

iterator

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

list

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

map

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

memory

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

numeric

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

queue

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

C and C++ Run-Time Libraries Guide

set

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

stack

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

utility

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

vector

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

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

fstream.h

The `fstream.h` header defines several `iostreams` template classes that manipulate external files.

iomanip.h

The `iomanip.h` header declares several `iostreams` manipulators that take a single argument.

iostream.h

The `iostream.h` header declares the `iostreams` objects that manipulate the standard streams.

new.h

The `new.h` header declares several functions that allocate and free storage.

Documented Library Functions

The following tables list the library functions documented in this chapter.

 These tables list the functions by the header file in which they are located, whereas the [“C Run-Time Library Reference” on page 2-24](#) presents the functions in alphabetic order.

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

“isalnum”	“isalpha”	“isctrl”
“isdigit”	“isgraph”	“islower”
“isprint”	“ispunct”	“isspace”
“isupper”	“isxdigit”	“tolower”
“toupper”		

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

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

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

“longjmp”	“setjmp”
---------------------------	--------------------------

Documented Library Functions

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

<code>“clear_interrupt”</code>	<code>“interrupt”</code>	<code>“raise”</code>
<code>“signal”</code>		

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

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

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

<code>clearerr</code>	<code>fclose</code>	<code>feof</code>
<code>ferror</code>	<code>fflush</code>	<code>fgetc</code>
<code>fgets</code>	<code>fprintf</code>	<code>fputc</code>
<code>fputs</code>	<code>fopen</code>	<code>fread</code>
<code>freopen</code>	<code>fscanf</code>	<code>fwrite</code>
<code>getc</code>	<code>getchar</code>	<code>gets</code>
<code>perror</code>	<code>putc</code>	<code>putchar</code>
<code>puts</code>	<code>scanf</code>	<code>setbuf</code>
<code>setvbuf</code>	<code>sprintf</code>	<code>sscanf</code>
<code>ungetc</code>	<code>vfprintf</code>	<code>vprintf</code>
<code>vsprintf</code>		

Table 2-10. Documented 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>

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

<code>"strtod"</code>	<code>"strtodf"</code>	<code>"strtol"</code>
<code>"strtol"</code>		

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

<code>"memchr"</code>	<code>"memcmp"</code>	<code>"memcpy"</code>
<code>"memcpy_from_shared"</code>	<code>"memcpy_to_shared"</code>	<code>"memmove"</code>
<code>"memset"</code>	<code>"strncat"</code>	<code>"strchr"</code>
<code>"strcmp"</code>	<code>"strcoll"</code>	<code>"strcpy"</code>
<code>"strcspn"</code>	<code>"strerror"</code>	<code>"strlen"</code>
<code>"strncat"</code>	<code>"strncmp"</code>	<code>"strncpy"</code>
<code>"strpbrk"</code>	<code>"strrchr"</code>	<code>"strspn"</code>
<code>"strstr"</code>	<code>"strtok"</code>	<code>"strxfrm"</code>

C Run-Time Library Reference

The C run-time library is a collection of functions that you can call from your C programs.



The information that follows applies to all of the functions in the library.

Notation Conventions

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

Reference Format

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

Name and Purpose of the function

Synopsis—Required header file and functional prototype

Description—Function specification

Error Conditions—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 signal. If the SIGABRT handler returns, `abort()` calls `exit()` to terminate the program with a failure condition.

Error Conditions

The `abort` function does not return.

Example

```
#include <stdlib.h>
extern int errors;

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

See Also

[atexit](#),

abs

absolute value

Synopsis

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

Description

The `abs` function returns the absolute value of its `int` input. The `abs` function is implemented through a built-in. The built-in causes the compiler to emit an inline instruction to perform the required function at the point where `abs` is called.

 `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 the argument. The input must be in the range $[-1, 1]$. The output, in radians, is in the range $[0, \pi]$.

The `acos_fr16` function is only defined for input values between 0 and 0.9 ($=0x7333$). The input argument is in radians. Output values range from $\text{acos}(0)*2/\pi$ ($=0x7FFF$) to $\text{acos}(0.9)*2/\pi$ ($=0x24C1$).

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 acos_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 $-\pi/2$ to $\pi/2$.

The `asin_fr16` function is only defined for input values between -0.9 (=0x8ccd) and 0.9 (=0x7333). The input argument is in radians. Output values range from $\text{asin}(-0.9) \cdot 2/\pi$ (=0xa4C1) to $\text{asin}(0.9) \cdot 2/\pi$ (=0x5B3F).

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 $-\pi/2$ to $\pi/2$.

The `atan_fr16` function covers the output range from $-\pi/4$ (input value 0x8000, output value 0x9B78) to $\pi/4$ (input value 0x7FFF, output value 0x6488). The input argument is in radians.

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 (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 $[-\pi, \pi]$.

The `atan2_fr16` function uses the full range from $-\pi/4$ to $\pi/4$ (0x8000 to 0x7FFF) for both input and output arguments. This corresponds to a scaling by π compared to the floating-point function. The input argument is in radians.

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 = p/2 */
```

See Also

[atan](#),

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

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

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

atol

convert string to long integer

Synopsis

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

Description

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



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

Error Conditions

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

Example

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

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

See Also

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

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 pre-sorted array, where:

- `key` is a pointer 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 as parameters a pointer to the key and a pointer to an array element and 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 if the `key` is not found in the array.

C Run-Time Library Reference

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 nmem, 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);
float ceilf(float)
```

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](#)

clear_interrupt

clear a pending signal

Synopsis

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

Description

The `clear_interrupt` function clears the signal `sig` in the `IRPTL` register. The `sig` argument must be one of the processor signals shown below for the ADSP-219x DSPs.

Table 2-12. ADSP-219x Signals

Sig Value	Definition
SIG_PWRDWN	power down interrupt
SIG_STACKINT	PC, LOOP, or COUNTER overflow on push, or on pop when empty
SIG_KERNEL	kernel interrupt
SIG_INT4	user-assignable
SIG_INT5	user-assignable
SIG_INT6	user-assignable
SIG_INT7	user-assignable
SIG_INT8	user-assignable
SIG_INT9	user-assignable
SIG_INT10	user-assignable
SIG_INT11	user-assignable
SIG_INT12	user-assignable
SIG_INT13	user-assignable

C Run-Time Library Reference

Error Conditions

The `clear_interrupt` function returns a 1 if the interrupt was pending, a -1 if the parameter is not a valid signal, or 0 is returned otherwise.

Example

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

See Also

[interrupt](#), [raise](#), [signal](#)

cos

cosine

Synopsis

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

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 is defined by the domain of `0x8000` to `0x7FFF` 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);
float coshf (float)
```

Description

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

Error Conditions

The `cosh` function returns the IEEE constant `+Inf` if the argument is outside the domain.

Example

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

y = cosh(x);
```

See Also

[sinh](#)

disable_interrupts

disable interrupts

Synopsis

```
#include <sysreg.h>
void disable_interrupts(void)
```

Description

The `disable_interrupts` function causes the compiler to emit an instruction to disable hardware interrupts.

The `disable_interrupts` function is implemented as a compiler built-in. The emitted instruction will be inlined at the point of `disable_interrupts` use. The inclusion of the `sysreg.h` include file is mandatory when using `disable_interrupts`.

The `disable_interrupts` function does not return a value.

Error Conditions

The `disable_interrupts` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>
main(){
    disable_interrupts();    // emits "DIS INTS;"
                           // instruction inline
}
```

See Also

[enable_interrupts](#), [io_space_read](#), [io_space_write](#), [mode_change](#),
[sysreg_read](#), [sysreg_write](#)

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](#)

enable_interrupts

enable interrupts

Synopsis

```
#include <sysreg.h>
void enable_interrupts(void)
```

Description

The `enable_interrupts` function causes the compiler to emit an instruction to enable hardware interrupts.

The `enable_interrupts` function is implemented as a compiler built-in. The emitted instruction will be inlined at the point of `enable_interrupts` use. The inclusion of the `sysreg.h` include file is mandatory when using `enable_interrupts`.

The `enable_interrupts` function does not return a value.

Error Conditions

The `enable_interrupts` function does not return, raise or set any error conditions.

Example

```
#include <sysreg.h>
main(){
    enable_interrupts();    // emits "ENA INTS;"
                           // instruction inline
}
```

See Also

[disable_interrupts](#), [io_space_read](#), [io_space_write](#), [mode_change](#),
[sysreg_read](#), [sysreg_write](#)

C Run-Time Library Reference

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 `AX1`, and control is passed to the label `__lib_prog_term`, which is defined in the run-time header file.

Error Conditions

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

Example

```
#include <stdlib.h>

exit(EXIT_SUCCESS);
```

See Also

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

exp

exponential

Synopsis

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

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` and stores the value `ERANGE` in `errno` when there is an overflow error. In the case of underflow, the `exp` function returns a zero.

Example

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

See Also

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

external_memory_read

read from external memory

Synopsis

```
#include <sysreg.h>
void external_memory_read(int, void*)
```

Description

The `external_memory_read` function causes the compiler to emit instructions to read from external memory at the address passed as two parameters and set the value read from that address as a return value. The first parameter is the value of the top eight bits of the 24-bit external memory address to read. The second parameter is the lower 16 bits of the address of the external memory to read.

This function is implemented as a compiler built-in. The emitted instructions will be inlined at the point of `external_memory_read` use. The inclusion of the `sysreg.h` include file is mandatory when using `external_memory_read`.

Error Conditions

The `external_memory_read` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>
section("external_memory_section")
static int GlobalTable[256];

int main() {
    int page, read_value;
    asm("%0 = PAGE(GlobalTable);" : "=e"(page): : );
```

```
read_value = external_memory_read(page, &GlobalTable[1]);  
return read_value;  
}
```

See Also

[external_memory_write](#), [disable_interrupts](#), [enable_interrupts](#),
[io_space_read](#), [io_space_write](#), [mode_change](#), [sysreg_read](#), [sysreg_write](#)

external_memory_write

write to external memory

Synopsis

```
#include <sysreg.h>
void external_memory_write(int, void*, int)
```

Description

The `external_memory_write` function causes the compiler to emit instructions to write to external memory at the address passed in the first two parameters with the value passed in the last parameter. The first parameter is the value of the top eight bits of the 24-bit external memory address to write. The second parameter is the lower 16-bits of the address of the external memory to write.

This function is implemented as a compiler built-in. The emitted instructions will be inlined at the point of `external_memory_write` use. The inclusion of the `sysreg.h` include file is mandatory when using `external_memory_write`.

Error Conditions

The `external_memory_write` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>
section("external_memory_section")
static int GlobalTable[256];

int main() {
    int page, value_to_write = 0;
    asm("%0 = PAGE(GlobalTable);" : "=e"(page): : );
```

```
    external_memory_write(page, &GlobalTable[1], value_to_write);  
}
```

See Also

[external_memory_read](#), [disable_interrupts](#), [enable_interrupts](#),
[io_space_read](#), [io_space_write](#), [sysreg_read](#), [sysreg_write](#)

fabs

float absolute value

Synopsis

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

Description

The `floor` function produces 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 first argument into the second 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](#), [ldiv](#), [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 *expptr);
float frexpf(float f, int *expptr);
```

Description

The `frexp` function separates a floating-point input into a normalized fraction and a (base 2) exponent. The function returns the first argument, a fraction 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);
void (*interruptf(int sig, void(*func)(int))) (int);
void (*interrupts(int sig, void(*func)(int))) (int);
```

Description

These functions are Analog Devices extensions to the ANSI standard.

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; the `signal` function executes the function only once.

The different variants of the `interrupt` functions differentiate between handler dispatching functions. The variants will be appropriate for some applications and provide improved efficiency. The default `interrupt` function dispatcher saves and restores all scratch registers and modes on the data stack around a call to the handler (`func`) when servicing an interrupt. This dispatcher will pass the interrupt ID (for example, `SIG_PWRDWN`) to the handler as its parameter.

The `interruptf` interrupt dispatcher is similar to `interrupt`, except that it switches between primary and secondary register sets to save and restore registers instead of using the data stack. The `interruptf` function cannot be used in applications where nested interrupts are enabled. This interrupt dispatcher will pass the interrupt ID to the handler as its parameter.

The `interrupts` interrupt dispatcher saves and restores only the smallest number of registers and modes required to determine if a handler has been registered and to call that handler. The handler passed as input to

C Run-Time Library Reference

interrupts must be declared using the `#pragma interrupt` directive (see [on page 1-100](#)). The `#pragma altregisters` directive (see [on page 1-101](#)) may be used in conjunction with the `interrupt pragma` in the definition of the handler. This dispatcher will *not* pass the interrupt ID to the handler.

The `sig` argument must be one of the signals listed in priority order in [Table 2-13](#).

Table 2-13. Interrupt Function Signals - Values and Meanings

Sig Value	Definition
SIG_PWRDWN	power down interrupt
SIG_STACKINT	PC, LOOP, or COUNTER overflow on push, or on pop when empty
SIG_KERNEL	kernel interrupt
SIG_INT4	user-assignable
SIG_INT5	user-assignable
SIG_INT6	user-assignable
SIG_INT7	user-assignable
SIG_INT8	user-assignable
SIG_INT9	user-assignable
SIG_INT10	user-assignable
SIG_INT11	user-assignable
SIG_INT12	user-assignable
SIG_INT13	user-assignable
SIGABRT	software interrupt
SIGILL	software interrupt
SIGINT	software interrupt
SIGSEGV	software interrupt

Table 2-13. Interrupt Function Signals - Values and Meanings (Cont'd)

Sig Value	Definition
SIGTERM	software interrupt
SIGFPE	software interrupt

The `interrupt` functions cause the receipt of the signal number `sig` to be handled in one of the following ways:

- `SIG_DFL`—The default action is taken.
- `SIG_IGN`—The signal is ignored.
- **Function Address**—The function pointed to by `func` is executed.

The function pointed to by `func` is executed each time the interrupt is received. The `interrupt` function must be called with the `SIG_IGN` argument to disable interrupt handling.



Interrupts are not nested by default.

Error Conditions

The `interrupt` functions return `SIG_ERR` and set `errno` equal to `SIG_ERR` if the requested interrupt is not recognized.

Example

```
include <signal.h>
void handler (int sig) { /* Interrupt Service Routine (ISR) */
}
main () {
    /* enable power down interrupt and register ISR */
    interrupt(SIG_PWRDWN, handler);

    /* disable power down interrupt */
    interrupt(SIG_PWRDWN, SIG_IGN);
}
```

C Run-Time Library Reference

```
/* enable power down interrupt and register ISR */  
interruptf(SIG_PWRDWN, handler);  
  
/* disable power down interrupt */  
interruptf(SIG_PWRDWN, SIG_IGN);  
}
```

See Also

[signal](#), [raise](#)

io_space_read

read I/O space

Synopsis

```
#include <sysreg.h>
int io_space_read(const int)
```

Description

The `io_space_read` function causes the compiler to emit an instruction to read from I/O space at the address passed as a parameter and set the value read from that address as a return value.

The `io_space_read` function is implemented as a compiler built-in. The emitted instruction will be inlined at the point of `io_space_read` use. The inclusion of the `sysreg.h` include file is mandatory when using `io_space_read`.

Error Conditions

The `io_space_read` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>
main(){
    int value = io_space_read(0xA);
}
```

See Also

[disable_interrupts](#), [enable_interrupts](#), [io_space_write](#), [mode_change](#), [sysreg_read](#), [sysreg_write](#)

io_space_write

write I/O space

Synopsis

```
#include <sysreg.h>
void io_space_write(const int, const int)
```

Description

The `io_space_write` function causes the compiler to emit an instruction to write to I/O space at the address passed as the first parameter the value passed as the second parameter.

The `io_space_write` function is implemented as a compiler built-in. The emitted instruction will be inlined at the point of `io_space_write` use. The inclusion of the `sysreg.h` include file is mandatory when using `io_space_write`.

The `io_space_write` function does not return a value.

Error Conditions

The `io_space_write` function does not return, raise or set any error conditions.

Example

```
#include <sysreg.h>
main(){
    io_space_write(0xA, 0xF);
}
```

See Also

[disable_interrupts](#), [enable_interrupts](#), [io_space_read](#), [mode_change](#), [sysreg_read](#), [sysreg_write](#)

isalnum

detect alphanumeric character

Synopsis

```
#include <ctype.h>
int isalnum(int c);
```

Description

The `isalnum` function determines if the argument is an alphanumeric character (A-Z, a-z, or 0-9). If the argument is not alphanumeric, `isalnum` returns a zero. If the argument is alphanumeric, `isalnum` returns a non-zero value.

Error Conditions

The `isalnum` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%3s", isalnum(ch) ? "alphanumeric" : "");
    putchar('\n');
}
```

See Also

[isalpha](#), [isdigit](#)

isalpha

detect alphabetic character

Synopsis

```
#include <ctype.h>
int isalpha(int c);
```

Description

The `isalpha` function determines if the input is an alphabetic character (A-Z or a-z). If the input is not alphabetic, `isalpha` returns a zero. If the input is alphabetic, `isalpha` returns a non-zero value.

Error Conditions

The `isalpha` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%2s", isalpha(ch) ? "alphabetic" : "");
    putchar('\n');
}
```

See Also

[isdigit](#), [isalnum](#)

isctrl

detect control character

Synopsis

```
#include <ctype.h>
int isctrl(int c);
```

Description

The `isctrl` function determines if 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 non-zero 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 if the input character is a decimal digit (0-9). If the input is not a digit, `isdigit` returns a zero. If the input is a digit, `isdigit` returns a non-zero value.

Error Conditions

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

Example

```
#include <ctype.h>
int ch;
for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%2s", isdigit(ch) ? "digit" : "");
    putchar('\n');
}
```

See Also

[isalpha](#), [isalnum](#), [isxdigit](#)

isgraph

detect printable character, not including white space

Synopsis

```
#include <ctype.h>
int isgraph(int c);
```

Description

The `isgraph` function determines if the argument is a printable character, not including a white space (0x21-0x7e). If the argument is not a printable character, `isgraph` returns a zero. If the argument is a printable character, `isgraph` returns a non-zero value.

Error Conditions

The `isgraph` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%2s", isgraph(ch) ? "graph" : "");
    putchar('\n');
}
```

See Also

[isalnum](#), [iscntrl](#), [isprint](#)

isinf

test for infinity

Synopsis

```
#include <math.h>
int isinff(float x);
int isinf(double x);
```

Description

The `isinff` function returns a zero if the argument is not set to the IEEE constant for +Infinity or -Infinity; otherwise, the function will return a non-zero value.

Error Conditions

The `isinf` function does not return or set any error conditions.

Example

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

static int fail=0;

main(){
    /* test int isinf(double) */
    union {
        double d; float f; unsigned long l;
    } u;

    #ifdef __DOUBLES_ARE_FLOATS__
        u.l=0xFF800000L; if ( isinf(u.d)==0 ) fail++;
        u.l=0xFF800001L; if ( isinf(u.d)!=0 ) fail++;
```

```
        u.l=0x7F800000L; if ( isinf(u.d)==0 ) fail++;
        u.l=0x7F800001L; if ( isinf(u.d)!=0 ) fail++;
    #endif

    /* test int isinff(float) */
    u.l=0xFF800000L; if ( isinff(u.f)==0 ) fail++;
    u.l=0xFF800001L; if ( isinff(u.f)!=0 ) fail++;
    u.l=0x7F800000L; if ( isinff(u.f)==0 ) fail++;
    u.l=0x7F800001L; if ( isinff(u.f)!=0 ) fail++;

    /* print pass/fail message */
    if ( fail==0 )
        printf("Test passed\n");
    else
        printf("Test failed: %d\n", fail);
}
```

See Also

[isnan](#)

islower

detect lowercase character

Synopsis

```
#include <ctype.h>
int islower(int c);
```

Description

The `islower` function determines if the argument is a lowercase character (a-z). If the argument is not lowercase, `islower` returns a zero. If the argument is lowercase, `islower` returns a non-zero value.

Error Conditions

The `islower` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%2s", islower(ch) ? "lowercase" : "");
    putchar('\n');
}
```

See Also

[isalpha](#), [isalpha](#), [isupper](#)

isnan

test for not-a-number (NaN)

Synopsis

```
#include <math.h>
int isnanf(float x);
int isnan(double x);
```

Description

The `isnan` function returns a zero if the argument is not set to an IEEE NaN (Not a Number); otherwise, the function will return a non-zero value.

Error Conditions

The `isnan` function does not return or set any error conditions.

Example

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

static int fail=0;

main(){
    /* test int isnan(double) */
    union {
        double d; float f; unsigned long l;
    } u;

    #ifdef __DOUBLES_ARE_FLOATS__
        u.l=0xFF800000L; if ( isnan(u.d)!=0 ) fail++;
        u.l=0xFF800001L; if ( isnan(u.d)==0 ) fail++;
        u.l=0x7F800000L; if ( isnan(u.d)!=0 ) fail++;
```

C Run-Time Library Reference

```
        u.l=0x7F800001L; if ( isnan(u.d)==0 ) fail++;
    #endif

    /* test int isnanf(float) */
    u.l=0xFF800000L; if ( isnanf(u.f)!=0 ) fail++;
    u.l=0xFF800001L; if ( isnanf(u.f)==0 ) fail++;
    u.l=0x7F800000L; if ( isnanf(u.f)!=0 ) fail++;
    u.l=0x7F800001L; if ( isnanf(u.f)==0 ) fail++;

    /* print pass/fail message */
    if ( fail==0 )
        printf("Test passed\n");
    else
        printf("Test failed: %d\n", fail);
}
```

See Also

[isinf](#)

isprint

detect printable character

Synopsis

```
#include <ctype.h>
int isprint(int c);
```

Description

The `isprint` function determines if the argument is a printable character (0x20-0x7E). If the argument is not a printable character, `isprint` returns a zero. If the argument is a printable character, `isprint` returns a non-zero value.

Error Conditions

The `isprint` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%3s", isprint(ch) ? "printable" : "");
    putchar('\n');
}
```

See Also

[isgraph](#), [isspace](#)

ispunct

detect punctuation character

Synopsis

```
#include <ctype.h>
int ispunct(int c);
```

Description

The `ispunct` function determines if the argument is a punctuation character. If the argument is not a punctuation character, `ispunct` returns a zero. If the argument is a punctuation character, `ispunct` returns a non-zero value.

Error Conditions

The `ispunct` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%3s", ispunct(ch) ? "punctuation" : "");
    putchar('\n');
}
```

See Also

[isalnum](#)

isspace

detect whitespace character

Synopsis

```
#include <ctype.h>
int isspace(int c);
```

Description

The `isspace` function determines if the argument is a blank space character (0x09-0x0D or 0x20). This includes space, form feed (`\f`), new line (`\n`), carriage return (`\r`), horizontal tab (`\t`) and vertical tab (`\v`).

If the argument is not a blank space character, `isspace` returns a zero. If the argument is a blank space character, `isspace` returns a non-zero value.

Error Conditions

The `isspace` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%2s", isspace(ch) ? "space" : "");
    putchar('\n');
}
```

See Also

[iscntrl](#), [isgraph](#)

isupper

detect uppercase character

Synopsis

```
#include <ctype.h>
int isupper(int c);
```

Description

The `isupper` function determines if the argument is an uppercase character (A-Z). If the argument is not an uppercase character, `isupper` returns a zero. If the argument is an uppercase character, `isupper` returns a non-zero value.

Error Conditions

The `isupper` function does not return any error conditions.

Example

```
#include <ctype.h>
int ch;

for (ch=0; ch<=0x7f; ch++) {
    printf("%#04x", ch);
    printf("%2s", isupper(ch) ? "uppercase" : "");
    putchar('\n');
}
```

See Also

[isalpha](#), [islower](#)

isxdigit

detect hexadecimal digit

Synopsis

```
#include <ctype.h>
int isxdigit(int c);
```

Description

The `isxdigit` function determines if the argument 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](#), [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 the second argument `n` to the exponent of the first argument `x`.

Error Conditions

If the result overflows, `ldexp` returns a NaN. If the result underflows, `ldexp` returns a zero.

Example

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

y = ldexp(0.5, 2);    /* y = 2.0 */
```

See Also

[exp](#), [pow](#)

C Run-Time Library Reference

ldiv

long division

Synopsis

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Description

The `ldiv` function divides `numer` by `denom`, and returns a structure of type `ldiv_t`. The type `ldiv_t` is defined as:

```
typedef struct {
    long int quot;
    long int rem;
} ldiv_t
```

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

$$\text{result.quot} * \text{denom} + \text{result.rem} = \text{numer}$$

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);
```

Description

The `log` function computes the natural (base *e*) logarithm of its input.

Error Conditions

The `log` function returns a zero and sets `errno` to `EDOM` if the input value is 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);
float log10f(float);
```

Description

The `log10` function returns the base 10 logarithm of its input.

Error Conditions

The `log10` function indicates a domain error (sets `errno` to `EDOM`) and returns a zero if the input is 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 (with the same `jmp_buf` argument).

The `longjmp` function takes as its arguments a jump buffer that contains the context at the time of the original `setjmp`. It also takes an integer, `return_val`, which `setjmp` returns if `return_val` is non-zero. Otherwise, `setjmp` returns a 1.

If `env` was not initialized through a previous call to `setjmp` or the function that called `setjmp` has since returned, the behavior is undefined. 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 <setjmp.h>
jmp_buf env;

if (setjmp(env) != 0) {
    printf("Unexpected return from subroutine\n");
}
```

C Run-Time Library Reference

```
        exit(EXIT_FAILURE);
    }

    ... /* intervening code */
    if(unexpected_error)
        longjmp(env,1);
```

See Also

[setjmp](#)

malloc

allocate memory

Synopsis

```
#include <stdlib.h>
void *malloc(size_t size);
```

Description

The `malloc` function returns a pointer to a block of memory of length `size`. The block of memory is uninitialized.

Error Conditions

The `malloc` function returns a null pointer if it is unable to allocate the requested memory.

Example

```
#include <stdlib.h>
int *ptr;

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

See Also

[calloc](#), [free](#), [realloc](#)

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 greater than the `s2` object. If the `s2` object is greater than the `s1` object, a negative value is returned. A zero is returned 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";
result=memcpy (b, a, 3);    /* *b="SRC" */
```

See Also

[memmove](#), [memcpy_from_shared](#), [memcpy_to_shared](#), [strcpy](#), [strncpy](#)

memcpy_from_shared

copy characters from an object in the ADSP-2192-12 processor's shared memory (0x20000-0x20FFF) to non-shared memory in default data area.

Synopsis

```
#include <string.h>
void *memcpy_from_shared(void *s1, void *s2, size_t n);
```

Description

The `memcpy_from_shared` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`.

The `s2` parameter must be an object in the ADSP-2192-12 processor's shared memory (0x20000-0x20FFF).

Error Conditions

The `memcpy_from_shared` function does not return, raise, or set any error conditions.

See Also

[memcpy](#), [memcpy_to_shared](#)

memcpy_to_shared

copy characters from object in default data area to an object in the ADSP-2192-12 processor's shared memory (0x20000-0x20FFF).

Synopsis

```
#include <string.h>
void *memcpy_to_shared(void *s1, void *s2, size_t n);
```

Description

The `memcpy_to_shared` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`.

The `s1` parameter must be an object in the ADSP-2192-12 processor's shared memory (0x20000-0x20FFF).

Error Conditions

The `memcpy_to_shared` function does not return, raise, or set any error conditions.

See also

[memcpy](#), [memcpy_from_shared](#)

memmove

move characters from one object to another

Synopsis

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Description

The `memmove` function moves `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The entire object is moved 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(str, 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](#), [memmove](#)

mode_change

change selected system modes

Synopsis

```
#include <sysreg.h>
void mode_change(const int);
```

Description

The `mode_change` function causes the compiler to emit instructions to enable and disable a series of modes using the zero latency mode control instructions.

The `mode_change` function takes as a parameter a constant integer bitmask that the compiler converts into a series of enable and disable mode settings.

The `sysreg.h` include file defines each mode bit setting value as a symbolic variable that can be used as a parameter to `mode_change`. These definitions are:

```
__MODE_ENA_AV_LATCH =0x1,
__MODE_ENA_AR_SAT   =0x2,
__MODE_ENA_M_MODE    =0x4,
__MODE_ENA_TIMER     =0x8,
__MODE_ENA_INT       =0x10,
__MODE_DIS_AV_LATCH  =0x100,
__MODE_DIS_AR_SAT    =0x200,
__MODE_DIS_M_MODE    =0x400,
__MODE_DIS_TIMER     =0x800,
__MODE_DIS_INT       =0x1000,
```

The `mode_change` function is implemented as a compiler built-in and the emitted instructions will be inlined at the point of `mode_change` use.

C Run-Time Library Reference

The inclusion of the `sysreg.h` include file is mandatory when using `mode_change`.

Error Conditions

The `mode_change` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>
main(){
    /* enable TIMER and disable AR saturation */
    mode_change( __MODE_ENA_TIMER | __MODE_DIS_AR_SAT );
}
```

See Also

[disable_interrupts](#), [enable_interrupts](#), [io_space_read](#), [io_space_write](#),
[sysreg_read](#), [sysreg_write](#)

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, double);
float powf(float, float);
```

Description

The `pow` function computes the value of the first argument raised to the power of the second argument.

Error Conditions

A domain error occurs if the first argument is negative and the second argument cannot be represented as an integer. If the first argument is zero, the second argument is less than or equal to zero, and the result cannot be represented, `EDOM` is stored in `errno`.

Example

```
#include <math.h>
double z;
z = pow(4.0, 2.0);    /* z = 16.0 */
```

See Also

[frexp](#)

qsort

quicksort

Prototype

```
#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`. The size of each object is specified by `size`.

The contents of the array are sorted into ascending order according to a comparison function pointed to by `compare`, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two elements compare as equal, their order in the sorted array is unspecified. The `qsort` function executes a binary search operation on a 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

C Run-Time Library Reference

- `compare` points to the function used to compare two elements. It takes as parameters a pointer to the key and a pointer to an array element.

It 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.

Return Value

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

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 2-14](#)

Table 2-14. Raise Function Signals - Values and Meanings

Sig Value	Definition
SIG_PWRDWN	power down interrupt
SIG_STACKINT	PC, LOOP, or COUNTER overflow on push, or on pop when empty
SIG_KERNEL	kernel interrupt
SIG_INT4	user-assignable
SIG_INT5	user-assignable
SIG_INT6	user-assignable
SIG_INT7	user-assignable
SIG_INT8	user-assignable
SIG_INT9	user-assignable
SIG_INT10	user-assignable
SIG_INT11	user-assignable
SIG_INT12	user-assignable
SIG_INT13	user-assignable

Table 2-14. Raise Function Signals - Values and Meanings (Cont'd)

Sig Value	Definition
SIGABRT	software interrupt
SIGILL	software interrupt
SIGINT	software interrupt
SIGSEGV	software interrupt
SIGTERM	software interrupt
SIGFPE	software interrupt

 Interrupts are *not* nested by default.

Error Conditions

The `raise` function returns a zero if successful, a non-zero value if it fails.

Example

```
#include <signal.h>
raise(SIGABRT);
```

See Also

[interrupt](#), [signal](#)

rand

random number generator

Synopsis

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

Description

The `rand` function returns a pseudo-random integer value in the range $[0, 2^{15} - 1]$.

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 $2^{15} - 1$.

Error Conditions

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

Example

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

i = rand();
```

See Also

[srand](#)

C Run-Time Library Reference

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

define a run-time label

Synopsis

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

The `setjmp` function saves the calling environment in the `jmp_buf` argument. The effect of the call is to declare a run-time label that can be jumped to via a subsequent call to `longjmp`.

When `setjmp` is called, it immediately returns with a result of zero to indicate that the environment has been saved in the `jmp_buf` argument. If, at some later point, `longjmp` is called with the same `jmp_buf` argument, `longjmp` will restore the environment from the argument. The execution will then resume at the statement immediately following the corresponding call to `setjmp`. The effect is as if the call to `setjmp` has returned for a second time but this time the function will return a non-zero result.

The effect of calling `longjmp` will be undefined if the function that called `setjmp` has returned in the interim.

Error Conditions

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

Example

```
#include <setjmp.h>
jmp_buf env;

if (setjmp(env) != 0) {
    printf("Unexpected return from subroutine\n");
}
```

C Run-Time Library Reference

```
        exit(EXIT_FAILURE);
    }
    ...    /* intervening code */
    if(unexpected_error)
        longjmp(env,1);
```

See Also

[longjmp](#)

signal

define signal handling

Synopsis

```
#include <signal.h>
void (*signal(int sig, void (*func)(int))) (int);
void (*signalf(int sig, void (*func)(int))) (int);
void (*signals(int sig, void (*func)(int))) (int);
```

Description

These functions are Analog Devices extensions to the ANSI standard.

The `signal` function determines how a signal received during program execution is handled. The signal functions cause a single execution the function pointed to by `func`; the `interrupt` functions cause the function to be executed for every interrupt.

The different variants of the `signal` functions differentiate between handler dispatching functions. The variants will be appropriate for some applications and provide improved efficiency. The default `signal` function dispatcher saves and restores all scratch registers and modes on the data stack around a call to the handler (`func`) when servicing an interrupt. This dispatcher will pass the interrupt ID (for example, `SIG_PWRDWN`) to the handler as its parameter.

The `signalf` interrupt dispatcher is similar to `interrupt`, except that it switches between primary and secondary register sets to save and restore registers instead of using the data stack. The `signalf` function cannot be used in applications where nested interrupts are enabled. This interrupt dispatcher will pass the interrupt ID to the handler as its parameter.

The `signals` interrupt dispatcher saves and restores only the smallest number of registers and modes required to determine if a handler has been registered and to call that handler. The handler passed as input to `signals`

C Run-Time Library Reference

must be declared using the `#pragma interrupt` directive (see [on page 1-100](#)). The `altregisters` directive (see [on page 1-101](#)) may be used in conjunction with the `interrupt pragma` in the definition of the handler. This dispatcher will *not* pass the interrupt ID to the handler.

The `sig` argument must be one of the signals listed in highest to lowest priority of interrupts in [Table 2-15](#).

Table 2-15. Signal Function Signals - Values and Meanings

Sig Value	Definition
SIG_PWRDWN	power down interrupt
SIG_STACKINT	PC, LOOP, or COUNTER overflow on push, or on pop when empty
SIG_KERNEL	kernel interrupt
SIG_INT4	user-assignable
SIG_INT5	user-assignable
SIG_INT6	user-assignable
SIG_INT7	user-assignable
SIG_INT8	user-assignable
SIG_INT9	user-assignable
SIG_INT10	user-assignable
SIG_INT11	user-assignable
SIG_INT12	user-assignable
SIG_INT13	user-assignable
SIGABRT	software interrupt
SIGILL	software interrupt
SIGINT	software interrupt
SIGSEGV	software interrupt
SIGTERM	software interrupt
SIGFPE	software interrupt

The `signal` function causes the receipt of the signal number `sig` to be handled in one of the following ways:

- `SIG_DFL`—The default action is taken.
- `SIG_IGN`—The signal is ignored.
- Function address—The function pointed to by `func` is executed. The function pointed to by `func` is executed once when the signal is received. Handling is then returned to the default state.



Interrupts are not nested by default.

Error Conditions

The `signal` function returns `SIG_ERR` and sets `errno` to `SIG_ERR` if it does not recognize the requested signal.

Example

```
#include <signal.h>
void handler (int sig) {    /* Interrupt Service Routine (ISR) */
}
main () {

    /* enable power down interrupt and register ISR */
    signal(SIG_PWRDWN, handler);

    /* disable power down interrupt */
    signal(SIG_PWRDWN, SIG_IGN);

    /* enable power down interrupt and register ISR */
    signal(SIG_PWRDWN, handler);

    /* disable power down interrupt */
    signal(SIG_PWRDWN, SIG_IGN);
}
```

See Also

[interrupt](#), [raise](#)

sin

sine

Synopsis

```
#include <math.h>
double sin(double x);
float sinf (float x);
fract16 sin_fr16 (fract16 x);
```

Description

The `sin` function returns the sine of the argument. The input is interpreted as a radian; the output is in the range $[-1, 1]$.

The `sin_fr16` function is defined by the domain of `0x8000` to `0x7FFF` corresponding to $[-\pi/2, \pi/2]$. This domain only covers half a cycle. Because of the periodic nature of the function, this is deemed sufficient to derive a full period if required. The input argument is in radians.

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 the argument `x`.

Error Conditions

The `sinh` function returns the IEEE constant `+Inf` 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 the argument.

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 is used to set the seed value for the `rand` function. A particular seed value always produces the same sequence of pseudo-random numbers.

Error Conditions

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

Example

```
#include <stdlib.h>

srand(22);
```

See Also

[rand](#)

strcat

concatenate strings

Synopsis

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

Description

The `strcat` function appends a copy of the null-terminated string pointed to by `s2` to the end of the null-terminated string pointed to by `s1`. It returns a pointer to the new `s1` string, which is null-terminated. The behavior of `strcat` is undefined if the two strings overlap.

Error Conditions

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

Example

```
#include <string.h>
char string1[50];
    string1[0] = 'A';
    string1[1] = 'B';
    string1[2] = '\0';
    strcat(string1, "CD");    /* new string is "ABCD" */
```

See Also

[strncat](#)

strchr

find first occurrence of character in string

Synopsis

```
#include <string.h>
char *strchr(const char *s1, int c);
```

Description

The `strchr` function returns a pointer to the first location in `s1`, a null-terminated string, that contains the character `c`.

Error Conditions

The `strchr` function returns a null 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 with 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 locale macro, `LC_COLLATE`. Because only the C locale is defined in the ADSP-219x DSP 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

[strrchr](#), [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 <string.h>
char *ptr1, *ptr2;
size_t len;

ptr1 = "Tried and Tested";
ptr2 = "aeiou";
len = strcspn (ptr1, ptr2);    /* len = 2 */
```

See Also

[strlen](#), [strspn](#)

strerror

get string containing error message

Synopsis

```
#include <string.h>
char *strerror(int errnum);
```

Description

The `strerror` function returns a pointer to a string containing an error message by mapping the number in `errnum` to that string.

Error Conditions

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

Example

```
#include <string.h>
char *ptr1;

ptr1 = strerror(1);
```

See Also

No 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 null-terminating 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`. It returns a pointer to the new `s1` string.

The behavior of `strncat` is undefined if the two strings overlap. The new `s1` string is terminated with a null (`'\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 with up to `n` characters of the null-terminated strings pointed to by `s1` and `s2`. It returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

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

Example

```
#include <string.h>
char *ptr1;
ptr1 = "TEST1";

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

See Also

[memcmp](#), [strcmp](#)

strncpy

copy characters from one string to another

Synopsis

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

Description

The `strncpy` function copies up to `n` characters of the null-terminated string pointed to by `s2` into the space pointed to by `s1`. If the last character copied from `s2` is not a NULL, the result will not end with a NULL.

The behavior of `strncpy` is undefined if the two objects overlap. The `strncpy` function returns the new `s1`. If the `s2` string contains fewer than `n` characters, the `s1` string is padded with null characters 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](#)

strchr

find last 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 last occurrence of character `c` in the null-terminated input string `s1`.

Error Conditions

The `strchr` function returns a null pointer if `c` is not found.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strchr(ptr1, "T");
/* ptr2 points to the second T of TESTING */
```

See Also

[memchr](#), [strchr](#)

strspn

return 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 pointed to by `s1` of the characters in the string pointed to by `s2`. This excludes the terminating null character in `s1`.

Error Conditions

If the string is not found, `strstr` returns a null pointer. If `s2` points to a string of zero length, `s1` is returned.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strstr(ptr1, "'E'");
/* ptr2 points to the E in TESTING */
```

See Also

[strchr](#)

strtod

convert portion of string to double representation

Synopsis

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr)
```

Description

The `strtod` function extracts a value from the string pointed to by `nptr`, and returns the value as a double. The `strtod` function expects `nptr` to point to a string of the form

`[whitespace] [sign] [digits] [.digits] [{d|D|e|E}[sign]digits]`

The `whitespace` token 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-

C Run-Time Library Reference

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

strtodf

convert portion of string to float representation

Synopsis

```
#include <stdlib.h>
float strtodf(const char *nptr, char **endptr)
```

Description

The `strtodf` function extracts a value from the string pointed to by `nptr`, and returns the value as a `float`. The `strtodf` function expects `nptr` to point to a string of the form

`[whitespace] [sign] [digits] [.digits] [{d|D|e|E}[sign]digits]`

The `whitespace` token 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 `strtodf` 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-

C Run-Time Library Reference

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;
float f;

f = strtodf ("2345.5E4 abc",&rem);
/* f = 2.3455E+7 ,rem = "abc" */
```

See Also

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

strtok

convert string to tokens

Synopsis

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

Description

The `strtok` function returns successive tokens from the string `s1`, where each token is delimited by characters from `s2`.

A call to `strtok`, with `s1` not NULL, returns a pointer to the first token in `s1`, where a token is a consecutive sequence of characters not in `s2`. `s1` is modified in place to insert a null character at the end of the token returned. If `s1` consists entirely of characters from `s2`, NULL is returned.

Subsequent calls to `strtok` with `s1` equal to NULL will 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, in which case the remainder of the string is tokenized 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" */
```

C Run-Time Library Reference

```
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. `ERANGE` is stored in `errno` in the case of either overflow or underflow.

C Run-Time Library Reference

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.

C Run-Time Library Reference

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](#), [strtod](#), [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`, and copies no more than `n` characters of the transformed string into the string pointed to by `s1`. The resultant string will include the terminating null character.

The function returns the length of the transformed string (not including the terminating null character). If `n` is zero and `s1` is set to the null pointer, then `strxfrm` will return the number of characters required for the transformed string. Overlapping strings are not supported



The transformation is such that `strcmp` will return the same result for two transformed strings as `strcoll` would for the same original strings. However, because only the C locale is defined in the ADSP-219x DSP environment, the `strxfrm` function is similar to the `strncpy` function except that the null character is always appended at the end of the output string.

Error Conditions

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

Example

```
#include <string.h>
char string1[50];
```

C Run-Time Library Reference

```
strxfrm(string1, "SOMEFUN", 49);  
/* SOMEFUN is copied into string1 */
```

See Also

[strcmp](#), [strcoll](#), [strncpy](#)

sysreg_read

read from non-memory-mapped register

Synopsis

```
#include <sysreg.h>
int sysreg_read(const int)
```

Description

The `sysreg_read` function causes the compiler to emit instructions to read the non-memory-mapped register passed as the first parameter and set the value read from that register as a return value.

The input parameter for `sysreg_read` can be one of the symbolic variable constants defined in `sysreg.h` or any valid 8-bit value used to represent a system control register.

The symbolic definitions in `sysreg.h` are:

```
/* General Register set */
sysreg_ASTAT = 0x0, // arithmetic status
sysreg_SSTAT = 0x1, // shifter status
sysreg_MSTAT = 0x2, // multiplier status
sysreg_ICNTL = 0x3, // interrupt control
sysreg_IMASK = 0x4, // interrupts enabled mask
sysreg_IRPTL = 0x5, // Interrupt Latch register
sysreg_DMPG1 = 0x6, // DMPG1 high address register
sysreg_DMPG2 = 0x7, // DMPG2 high address register
sysreg_IOPG = 0x8, // IOPG I/O page register
/* System Control Register set */
sysreg_B0 = 0x9, // B0 base register
sysreg_B1 = 0xa, // B1 base register
sysreg_B2 = 0xb, // B2 base register
sysreg_B3 = 0xc, // B3 base register
sysreg_B4 = 0xd, // B4 base register
```

C Run-Time Library Reference

```
sysreg_B5      = 0xe, // B5 base register
sysreg_B6      = 0xf, // B6 base register
sysreg_B7      = 0x10, // B7 base register
sysreg_SYSCTL  = 0x11, // SYSCTL register
sysreg_CACTL   = 0x12, // Cache Control register
```

The `sysreg_read` function is implemented as a compiler built-in and the emitted instructions will be inline at the point of `sysreg_read` use.

The inclusion of the `sysreg.h` include file is mandatory when using `sysreg_read`.

Error Conditions

The `sysreg_read` function does not return, raise, or set any error conditions. The compiler will not validate the input, instead it will rely on the assembler to fault erroneous values.

Example

```
#include <sysreg.h>
main(){
    int value = sysreg_read(sysreg_IMASK);
}
```

See Also

[disable_interrupts](#), [enable_interrupts](#), [io_space_read](#), [io_space_write](#), [mode_change](#), [sysreg_write](#)

sysreg_write

write to non-memory-mapped register

Synopsis

```
#include <sysreg.h>
void sysreg_write(const int, const unsigned int)
```

Description

The `sysreg_write` function causes the compiler to emit instructions to write the system register passed as a first parameter with the value passed as a second parameter.

The first parameter for `sysreg_write` can be one of the symbolic variable constants defined in `sysreg.h` or any valid 8-bit value used to represent a system control register. The second parameter is `unsigned`.

The symbolic definitions in `sysreg.h` are:

```
/* General Register set */
sysreg_ASTAT = 0x0, // arithmetic status
sysreg_SSTAT = 0x1, // shifter status
sysreg_MSTAT = 0x2, // multiplier status
sysreg_ICNTL = 0x3, // interrupt control
sysreg_IMASK = 0x4, // interrupts enabled mask
sysreg_IRPTL = 0x5, // Interrupt Latch register
sysreg_DMPG1 = 0x6, // DMPG1 high address register
sysreg_DMPG2 = 0x7, // DMPG2 high address register
sysreg_IOPG = 0x8, // IOPG I/O page register
/* System Control Register set */
sysreg_B0 = 0x9, // B0 base register
sysreg_B1 = 0xa, // B1 base register
sysreg_B2 = 0xb, // B2 base register
sysreg_B3 = 0xc, // B3 base register
sysreg_B4 = 0xd, // B4 base register
```

C Run-Time Library Reference

```
sysreg_B5      = 0xe,    // B5 base register
sysreg_B6      = 0xf,    // B6 base register
sysreg_B7      = 0x10,   // B7 base register
sysreg_SYSCTL  = 0x11,   // SYSCTL register
sysreg_CACTL   = 0x12,   // Cache Control register
```

The `sysreg_write` function is implemented as a compiler built-in. The emitted instructions will be inlined at the point of `sysreg_write` use.

The inclusion of the `sysreg.h` include file is mandatory when using `sysreg_write`.

Error Conditions

The `sysreg_write` function does not return, raise, or set any error conditions.

Example

```
#include <sysreg.h>
main(){
    sysreg_write(sysreg_IMASK,0x1);
}
```

See Also

[disable_interrupts](#), [enable_interrupts](#), [io_space_read](#), [io_space_write](#), [mode_change](#), [sysreg_read](#)

tan

tangent

Synopsis

```
#include <math.h>
double tan(double x);
float tanf(float x);
fractl6 tan_fr16(fractl6 x);
```

Description

The `tan` function returns the tangent of the argument `x`. The input, in radians, must be in the range `[-9099, 9099]`.

The `tan_fr16` function is only defined for input values between $-\pi/4$ (`=0x9B78`) and $\pi/4$ (`=0x6488`). The input argument is in radians. Output values range from `0x8000` to `0x7FFF`. The library function returns 0 for any input argument that is outside the defined domain.

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 the argument `x`.

Error Conditions

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

Example

```
#include <math.h>
double x,y;
y = tanh(x);
```

See Also

[cosh](#), [sinh](#)

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](#)

va_arg

get next argument in variable list

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 type parameter is a type name to which a `*` may be appended to obtain a pointer to an object of type type. 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;
```

C Run-Time Library Reference

```
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_end](#), [va_start](#)

va_end

reset variable list pointer

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 2-147.

See Also

[va_arg](#), [va_start](#)

va_start

set variable list pointer

Synopsis

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

Description

The `va_start` macro is used in a function declared to take a variable number of arguments. The macro initializes a pointer of type `va_list` which is used by `va_arg` to walk through the list of variable arguments.

The second argument is the name of the last named parameter in the function's parameter list; the list of variable arguments immediately follows this parameter. The `va_start` macro must be invoked before either the `va_arg` or `va_end` macro can be invoked.

Error Conditions

The `va_start` macro does not return an error condition.

Example

See “[va_arg](#)” on page 2-147.

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. These functions are in addition to the C/C++ run-time library functions that are described in Chapter 2, “C/C++ Run-Time Library”.

For more information on the algorithms on which many of the C library's math functions are based, see the Cody and Waite text “*Software Manual for the Elementary Functions*” from 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 cc219x compiler.
- “DSP Run-Time Library Reference” on page 3-21
It provides the complete reference for each DSP run-time library function provided with this release of the cc219x 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 on the following topics:

- [“Calling DSP Library Functions”](#)
- [“Linking DSP Library Functions” on page 3-3](#)
- [“Working with Library Source Code” on page 3-3](#)
- [“DSP Header Files” on page 3-4](#)

Calling DSP Library Functions

To use a DSP library function, call the function by name and give the appropriate arguments. The names and arguments for each function are described in the function's reference page in the section [“DSP Run-Time Library Reference” on page 3-21](#).

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

-  The function names are C function names. If you call C run-time library functions from an assembly language program, you must use the assembly version of the function name, which is the function name prefixed with an underscore. For more information on naming conventions, see the section, [“C/C++ and Assembly Language Interface” on page 1-123](#).
-  You can use the archiver, described in the *VisualDSP++ 3.0 Linker and Utilities Manual for ADSP-218x and ADSP-219x DSPs*, to build library archive files of your own functions.

Linking DSP Library Functions

When your 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 `219x\lib` directory, which is a subdirectory of the VisualDSP++ installation directory. The default Linker Description File (LDF) will do this automatically as it specifies that `libdsp.dlb` will be included in every link. If an application uses a customized LDF file, then either add `libdsp.dlb` to the customized LDF file, or alternatively use the compiler's `-ldsp` switch to specify that `libdsp.dlb` is to be added to the link line.

Working with Library Source Code

The source code for the functions in the DSP run-time library is provided with your VisualDSP++ software. By default, the installation program copies the source code to a subdirectory of the directory where the run-time libraries are kept, named `219x\lib\src\libdsp_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 run-time library functions, you can delete this directory and its contents to conserve disk space.

The source code is provided so you can customize specific functions for your own needs. To modify these files, you need proficiency in ADSP-219x DSP assembly language and an understanding of the run-time environment, as explained in [“C/C++ and Assembly Language Interface” on page 1-123](#). Before you make any modifications to the source code, copy the source code to a file with a different filename and rename the function itself. Test the function before you use it in your system to verify that it is functionally correct.



Analog Devices 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 cc219x compiler are:

- [“complex.h — Basic Complex Arithmetic Functions” on page 3-4](#)
- [“filter.h — DSP Filters and Transformations” on page 3-6](#)
- [“math.h — Math Functions” on page 3-10](#)
- [“matrix.h — Matrix Functions” on page 3-12](#)
- [“stats.h — Statistical Functions” on page 3-15](#)
- [“vector.h — Vector Functions” on page 3-16](#)
- [“window.h — Window Generators” on page 3-20](#)

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 the `complex.h` header file are listed in [Table 3-1 on page 3-5](#).

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 3-21](#).

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);

Table 3-1. Complex Functions (Cont'd)

Description	Prototype
Get Phase of Complex Number	<code>double arg (complex_double a);</code> <code>float argf (complex_float a);</code> <code>fract16 arg_fr16 (complex_fr16 a);</code>
Complex Conjugate	<code>complex_double conj (complex_double a);</code> <code>complex_float conjf (complex_float a);</code> <code>complex_fract16 conj_fr16 (complex_fract16 a);</code>
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>

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.

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.

The functions defined in the `filter.h` header file are listed in [Table 3-2](#) and [Table 3-3](#) and are described in detail in “[DSP Run-Time Library Reference](#)” on [page 3-21](#).

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 the FFT uses only half the number of points of the largest, the stride should be 2.

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. Transformation Functions

Description	Prototype
Fast Fourier Transform	
Generate FFT Twiddle Factor	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);

Table 3-3. Transformation Functions (Cont'd)

Description	Prototype
Generate FFT Twiddle Factors for 2-D FFT	void twidfft2d_fr16 (complex_fract16 w[], int n);
N Point Radix 2 Complex Input FFT	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);
N Point Radix 2 Real Input FFT	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);
N Point Radix 2 Inverse Input FFT	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);
N Point Radix 4 Complex Input FFT	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);
N Point Radix 4 Real Input FFT	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);
N Point Radix 4 Inverse Input FFT	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);
Nxn Point 2-D Complex Input FFT	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);

Table 3-3. Transformation Functions (Cont'd)

Description	Prototype
Nxn Point 2-D Real Input FFT	<pre>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);</pre>
Nxn Point 2-D Inverse FFT	<pre>void ifft2d_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>
Convolutions	
Convolution	<pre>void convolve_fr16 (const fract16 cin1[], int clen1, const fract16 cin2[], int clen2, fract16 cout[]);</pre>
2-D Convolution	<pre>void conv2d_fr16 (const fract16 *cin1, int crow1, int ccol1, const fract16 *cin2, int crow2, int ccol2, fract16 *cout);</pre>
2-D Convolution 3x3 Matrix	<pre>void conv2d3x3_fr16 (const fract16 *cin, int crow1, int ccol1, const fract16 cin2 [3] [3], fract16 *cout);</pre>
Compression/Expansion	
A-law Compression	<pre>void a_compress (const short in[], short out[], int n);</pre>
A-law Expansion	<pre>void a_expand (const short in[], short out[], int n);</pre>
μ-law Compression	<pre>void mu_compress (const short in[], short out[], int n);</pre>
μ-law Expansion	<pre>void mu_expand (const char in[], short out[], int n);</pre>

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 library reference section in Chapter 2, “[C/C++ Run-Time Library](#)”.

This header also provides prototypes for a number of additional math functions `clip`, `copysign`, `max`, `min`, and an integer function `countones`. These functions are described in “[DSP Run-Time Library Reference](#)” on [page 3-21](#).

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 (float 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>

Table 3-4. Math Library (Cont'd)

Description	Prototype
Hyperbolic Cosine	double cosh (double x); float coshf (float x);
Cotangent	double cot (double x); float cotf (float x);
Exponential	double exp (double x); float expf (float x);
Floor	double floor (double x); float 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);
Sine	double sin (double x); float sinf (float x); fract16 sin_fr16 (fract16 x);
Hyperbolic Sine	double sinh (double x); float sinh (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
Reciprocal of Square Root	double rsqrt (double x); float rsqrtf (float x);
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 definition of the complex types.

The matrix functions defined in the `matrix.h` header file are listed in [Table 3-5](#). 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, const double b, int n, int m, double *c); void matsaddf (const float *a, const float b, int n, int m, float *c); void matsadd_fr16 (const fract16 *a, const fract16 b, int n, int m, fract16 *c);</pre>
Real Matrix – Scalar Subtraction	<pre>void matssub (const double *a, const double b, int n, int m, double *c); void matssubf (const float *a, const float b, int n, int m, float *c); void matssub_fr16 (const fract16 *a, const 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>
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>

Table 3-5. Matrix Functions (Cont'd)

Description	Prototype
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>
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>
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>

Table 3-5. Matrix Functions (Cont'd)

Description	Prototype
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 “[DSP Run-Time Library Reference](#)” on [page 3-21](#).

Table 3-6. Stats 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>

Table 3-6. Stats Functions (Cont'd)

Description	Prototype
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 the definition of the complex types.

The functions in the `vector.h` header file are listed in [Table 3-7](#). In the prototypes in the table, `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 described by this header assume that input array arguments are constant—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.

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>
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>

Table 3-7. Vector Functions (Cont'd)

Description	Prototype
Maximum Value of Vector Elements	double vecmax (const double a [], int n); float vecmaxf (const float a [], int n); fract16 vecmax_fr16 (const fract 16 a [], int n);
Minimum Value of Vector Elements	double vecmin (const double a [], int n); float vecminf (const float a [], int n); fract16 vecmin_fr16 (const fract16 a [], int n);
Index of Maximum Value of Vector Elements	int vecmaxloc (const double a [], int n); int vecmaxlocf (const float a [], int n); int vecmaxloc_fr16 (const fract16 a [], int n);
Index of Minimum Value of Vector Elements	int vecminloc (const double a [], int n); int vecminlocf (const float a [], int n); int vecminloc_fr16 (const fract16 a [], int n);
Complex Vector + Scalar Addition	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);
Complex Vector – Scalar Subtraction	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);
Complex Vector * Scalar Multiplication	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);

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 Multiplication	<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>
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 [], const complex_fract16 c [], int n);</pre>

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 3-21](#).

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 3-8. Window Generator Functions

Description	Prototype
Generate Bartlett Window	<code>void gen_bartlett_fr16 (fract16 w[], int a, int n);</code>
Generate Blackman Window	<code>void gen_blackman_fr16 (fract16 w[], int a, int n);</code>
Generate Gaussian Window	<code>void gen_gaussian_fr16 (fract16 w[], float alpha, int a, int n);</code>
Generate Hamming Window	<code>void gen_hamming_fr16 (fract16 w[], int a, int n);</code>
Generate Hanning Window	<code>void gen_hanning_fr16 (fract16 w[], int a, int n);</code>
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.

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 the 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

The `a_compress` function takes a vector of linear 13-bit signed speech samples and performs A-law compression according to ITU recommendation G.711. Each sample is compressed to 8 bits and is returned in the vector pointed to by `out`.

Algorithm

```
C(k)=a-law compression of A(k)
for k=0 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

The `a_expand` function inputs a vector of 8-bit compressed speech samples and expands them according to ITU recommendation G.711. Each input value is expanded to a linear 13-bit signed sample in accordance with the A-law definition and is returned in the vector pointed to by `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 <code>argf()</code> , <code>arg()</code>
-1.0 to +1.0	for <code>arg_fr16()</code>

Note

$\text{Im}(a) / \text{Re}(a) \leq 1$	for <code>arg_fr16()</code>
--------------------------------------	------------------------------

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 a is the mean value of input vector a.

Domain

-3.4 x 10 ³⁸ to +3.4 x 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 (complex_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 to 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 two complex inputs: *a* and *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

-3.4×10^{38} to $+3.4 \times 10^{38}$ 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 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. The function does not impose any special memory alignment requirements on the arrays. However, benefits in run-time performance will be realized if the output array is allocated on an address boundary that is multiple of twice the FFT size. If the input data can be overwritten, then optimum memory usage can be achieved by specifying the input array as either the output array or as the temporary array. Specifying the input array as the temporary array will also result in increased run-time performance.

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 `cfft_fr16`, then the stride factor has to be set appropriately; otherwise it should be 1.

The argument `scale_method` controls how the function should scale the output to avoid overflow. If no scaling is selected by setting `scale_method` to zero, then the input signal should be sufficiently conditioned to avoid overflow. The `block_exponent` argument will be set to zero.

The function will perform static scaling if `scale_method` is set to 1. For static scaling, the function will scale intermediate results to prevent overflow. The final output will be scaled by $1/n$, and `block_exponent` will be set to $\log_2(n)$.

If `scale_method` is set to 2, then the function will select dynamic scaling. Under dynamic scaling, the function will inspect the intermediate results and will only scale to avoid overflow. Dynamic scaling therefore minimizes loss of precision but at the possible cost of slightly reduced performance. The `block_exponent` argument will be set to a value between 0 (which indicates that no scaling was performed) and $\log_2(n)$ (as if static scaling was performed).

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 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. The function does not impose any special memory alignment requirements on the arrays. However, benefits in run-time performance will be realized if the output array is allocated on an address boundary that is multiple of twice the FFT size. If the input data can be overwritten, then optimum memory usage can be achieved by specifying the input array as either the output array or as the temporary array. Specifying the input array as the temporary array will also result in increased run-time performance.

The twiddle table is passed in the argument `w`, which must contain at least $\frac{3}{4}n$ twiddle coefficients. The function `twidfftrad4_fr16` may be used to initialize the array. If the twiddle table contains more coefficients than needed for a particular call on `cffttrad4_fr16`, then the stride factor has to be set appropriately; otherwise it should be one.

The argument `scale_method` controls how the function should scale the output to avoid overflow. If no scaling is selected by setting `scale_method` to zero, then the input signal should be sufficiently conditioned to avoid overflow. The `block_exponent` argument will be set to zero.

The function will perform static scaling if `scale_method` is set to 1. For static scaling, the function will scale intermediate results to prevent overflow. The final output will be scaled by $1/n$, and `block_exponent` argument will be set to $\log_2(n)$.

If `scale_method` is set to 2, then the function will select dynamic scaling. Under dynamic scaling, the function will inspect the intermediate results and will only scale to avoid overflow. Dynamic scaling therefore minimizes loss of precision but at the possible cost of slightly reduced performance. The `block_exponent` argument will be set to a value between 0 (which indicates that no scaling was performed) and $\log_2(n)$ (as if static scaling was performed).

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 3-31](#) for more information.

Domain

Input sequence length n must be a power of four, and at least 16.

cfft2d

NxN 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;         /* scaling 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. The function does not impose any special memory alignment requirements on the arrays. However, benefits in run-time performance will be realized if the output array is allocated on an address boundary that is multiple of twice the FFT size.

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 `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. However, the current version of the function ignores the argument and always scales the output by $n*n$; this is equivalent to static scaling. The function will also set `block_exponent` to $\log_2(n)$.

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                          */
```

The function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    int k;                    /* Number of coefficients    */
    complex_fract16 *h;       /* Filter coefficients        */
    complex_fract16 *d;       /* Start of delay line       */
    complex_fract16 *p;       /* Read/write pointer        */
} cfir_state_fr16;
```

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`.

DSP Run-Time Library Reference

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:

```
#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

$$\text{Re}(c) = \text{Re}(a) * \text{Re}(b) - \text{Im}(a) * \text{Im}(b)$$
$$\text{Im}(c) = \text{Re}(a) * \text{Im}(b) + \text{Im}(a) * \text{Re}(b)$$

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_float b)
```

```
complex_double conj (complex_double a, complex_double b)
```

```
complex_fract16 conj_fr16 (complex_fract16 a, complex_fract16 b)
```

Description

This function conjugates the complex input *a* and returns the result.

Algorithm

$$\text{Re}(c) = \text{Re}(a)$$
$$\text{Im}(c) = -\text{Im}(a)$$

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 */
float 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`, then 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

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `conjf( )`, `conj( )`

-1.0 to 1.0 for `conj_fr16( )`

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      */
fract16 *mrow2;           /* number of rows in matrix 1    */
int mcol2;                /* number of columns in matrix 2 */
fract16 *mout;            /* pointer to output matrix       */
```

Description

This function computes the 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 $(mrow1 + mrow2 - 1) \times (mcol1 + mcol2 - 1)$.

Algorithm

Two dimensional input matrix `min1` is convolved with input matrix `min2`, placing the result in a matrix pointed to by `mout`.

$$mout[c, r] = \sum_{i=0}^{mcol2-1} \sum_{j=0}^{mrow2-1} min1[c+i, r+j] \bullet min2[(mcol2-1)-i, (mrow2-1)-j]$$

for `c = 0` to `mcol1+mcol2-1` and `r = 0` to `mrow2-1`

Domain

-1.0 to +1.0

conv2d3x3

2-D convolution

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 the two-dimensional convolution of matrix `min1` (size `mrow1` x `mcol1`) with matrix `min2` (size 3x3).

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

copy the sign

Synopsis

```
#include <math.h>
```

```
float copysignf (float parm1, float parm2)
```

```
double copysign (double parm1, double parm2)
```

```
fract16 copysign_fr16 (fract16 parm1, fract16 parm2)
```

Description

This function 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] for `cotf( )`, `cot( )`

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 word.

Algorithm

$$\text{return} = \sum_{j=0}^{j=N-1} \text{bit}[j] \text{ of word}$$

where N is the number of bits in a 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*, and stores the result to output vector *c*.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} (a_j - \bar{a}) * (b_{j+k} - \bar{b}) \right)$$

where $k=\{0,1,...,m-1\}$, $\bar{a}$ is the mean value of input vector *a* and $\bar{b}$ is the mean value of input vector *b*.

Domain

-3.4 x 10 ³⁸ to +3.4 x 10 ³⁸	for crosscohf()
-1.0 to +1.0	for crosscoh_fr16()

crosscorr

cross-correlation

Synopsis

```
#include <stats.h>

void crosscorrf(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,b,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, and 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

-3.4×10^{38} to $+3.4 \times 10^{38}$	for crosscorrf()
-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 <code>csubf()</code> , <code>csub()</code>
-1.0 to 1.0	for <code>csub_fr16()</code>

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 an FIR filter. The function generates the filtered response of the input data `x` and stores the result in the output vector `y`. The number of input samples and the length of the output vector is specified by the argument `n`.

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`, 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

$$y(i) = \sum_{j=0}^{k-1} h(j) * x(i-j) \quad \text{for } i = 0, 1, \dots, n-1$$

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 input samples         */
fir_state_fr16 *s;      /* Pointer to filter state structure */
```

The function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    fract16 *h;          /* Filter coefficients             */
    fract16 *d;          /* Start of delay line            */
    fract16 *p;          /* Read/Write pointer             */
    int k;               /* Number of coefficients         */
    int l;               /* Interpolation/decimation index */
} fir_state_fr16;
```

Description

The `fir_decima_fr16` function performs an 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 number of input samples is specified by the argument `n`, and the size of the output vector should be `n/l` where `l` is the decimation index.

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

$$y(i) = \sum_{j=0}^{k-1} x(i * l - j) * h(k-1+j)$$

where $i = 0, 1, \dots, (n/l) - 1$

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 */
```

The function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    fract16 *h;              /* Filter coefficients          */
    fract16 *d;              /* Start of delay line         */
    fract16 *p;              /* Read/Write pointer          */
    int k;                   /* Number of coefficients      */
    int l;                   /* Interpolation/decimation index */
} fir_state_fr16;
```

Description

The `fir_interp_fr16` function performs an 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 number of input samples is specified by the argument `n`, and the size of the output vector should be `n*l` where `l` is the interpolation index.

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 filter characteristics 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

$$y(l*i+m) = \sum_{j=0}^{k-1} x(i-j) * h((k-1+j)+m*k) \quad \text{for } m=0,1,\dots,l-1$$

where $i = 0,1,\dots,n-1$

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 of the window required is specified by the parameter N , and the stride parameter a is used to space the window values within the output vector w . The length of the output vector should therefore be $N*a$.

The Bartlett window is similar to the Triangle window but has the different properties:

- The Bartlett window always returns a window with two zeros on either end of the sequence, so that for odd n , the center section of an $N+2$ Bartlett window equals an N Triangle window.
- For even n , the Bartlett window is still the convolution of two rectangular sequences. There is no standard definition for the Triangle window for even n ; the slopes of the Triangle window are slightly steeper than those of the Bartlett window.

Algorithm

$$w[n] = 1 - \left| \frac{n - \frac{N-1}{2}}{\frac{N-1}{2}} \right|$$

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 of the window required is specified by the parameter N , and the stride parameter a is used to space the window values within the output vector w . The length of the output vector should therefore be $N*a$.

Algorithm

$$w[n] = 0.42 - 0.5 \cos\left(\frac{2\pi n}{N-1}\right) + 0.08 \cos\left(\frac{4\pi n}{N-1}\right)$$

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 of the window required is specified by the parameter N , and the stride parameter a is used to space the window values within the output vector w . The length of the output vector should therefore be $N*a$.

Algorithm

$$w(n) = \exp \left[-\frac{1}{2} \left(\alpha \frac{n - N/2 - 1/2}{N/2} \right)^2 \right]$$

where $n = \{0, 1, 2, \dots, N-1\}$ and a is an input parameter

Domain

$a > 0$; $N > 0$; $a > 0.0$

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 of the window required is specified by the parameter N , and the stride parameter a is used to space the window values within the output vector w . The length of the output vector should therefore be $N*a$.

Algorithm

$$w[n] = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right)$$

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 of the window required is specified by the parameter N , and the stride parameter a is used to space the window values within the output vector w . The length of the output vector should therefore be $N*a$. This window is also known as the Cosine window.

Algorithm

$$w[n] = 0.5 - 0.5 \cos\left(\frac{2\pi n}{N-1}\right)$$

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 of the window required is specified by the parameter N , and the stride parameter a is used to space the window values within the output vector w . The length of the output vector should therefore be $N*a$. This window is also known as the Blackman-Harris window.

Algorithm

$$w[n] = 0.35875 - 0.48829 * \cos\left(\frac{2\pi n}{N-1}\right) + 0.14128 * \cos\left(\frac{4\pi n}{N-1}\right) + 0.01168 * \cos\left(\frac{6\pi n}{N-1}\right)$$

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 of the window required is specified by the parameter N, and the stride parameter a is used to space the window values within the output vector w. The length of the output vector should therefore be N*a. The b value is specified by parameter beta.

Algorithm

$$w[n] = \frac{I_0 \left[\beta \left(1 - \left[\frac{n-\alpha}{\alpha} \right]^2 \right)^{1/2} \right]}{I_0(\beta)}$$

where n = {0, 1, 2, ..., N-1}, (= (N - 1) / 2, and I0(b) represents the zeroth-order modified Bessel function of the first kind.

Domain

a > 0; N > 0; b > 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 of the window required is specified by the parameter N , and the stride parameter a is used to space the window values within the output vector w . The length of the output vector should therefore be $N*a$.

Algorithm

$$w[n] = 1$$

where $n = \{0, 1, 2, \dots, N-1\}$

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 of the window required is specified by the parameter *N*, and the stride parameter *a* is used to space the window values within the output vector *w*.

Refer to the Bartlett window ([on page 3-58](#)) regarding the relationship between it and the Triangle window.

Algorithm

For even *n*, the following equation applies:

$$w[n] = \begin{cases} \frac{2n+1}{N} & n < N/2 \\ \frac{2N-2n-1}{N} & n > N/2 \end{cases}$$

where $n = \{0, 1, 2, \dots, N-1\}$

DSP Run-Time Library Reference

For odd n , the following equation applies:

$$w[n] = \begin{cases} \frac{2n+2}{N+1} & n < N/2 \\ \frac{2N-2n}{N+1} & n > N/2 \end{cases}$$

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 `gen_hanning window` ([on page 3-63](#)).

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,c,max,min,n,m)
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*.

Algorithm

It bins *n* elements of input vector *a* into *m* equally spaced containers, and returns the number of elements in each container.

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$	for histogramf()
-1.0 to +1.0	for histogram_fr16()

ifft

N point 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. The function does not impose any special memory alignment requirements on the arrays. However, benefits in run-time performance will be realized if the output array is allocated on an address boundary that is multiple of twice the FFT size. If the input data can be overwritten, then optimum memory usage can be achieved by 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.

DSP Run-Time Library Reference

The argument `scale_method` controls how the function should scale the output to avoid overflow. If no scaling is selected by setting `scale_method` to zero, then the input signal should be sufficiently conditioned to avoid overflow. The `block_exponent` argument will be set to zero.

The function will perform static scaling if `scale_method` is set to 1. For static scaling, the function will scale intermediate results to prevent overflow. The final output will be scaled by $1/n$, and `block_exponent` will be set to $\log_2(n)$.

If `scale_method` is set to 2, then the function will select dynamic scaling. Under dynamic scaling, the function will inspect the intermediate results and will only scale to avoid overflow. Dynamic scaling therefore minimizes loss of precision but at the possible cost of slightly reduced performance. The `block_exponent` argument will be set to a value between 0 (which indicates that no scaling was performed) and $\log_2(n)$ (as if static scaling was performed).

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 inverse 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. The function does not impose any special memory alignment requirements on the arrays. However, benefits in run-time performance will be realized if the output array is allocated on an address boundary that is multiple of twice the FFT size. If the input data can be overwritten, then optimum memory usage can be achieved by specifying the input array as the output array.

The twiddle table is passed in the argument *w*, which must contain at least $\frac{1}{4}n$ 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 `iffttrad4_fr16`, then the stride factor has to be set appropriately; otherwise it should be 1.

The argument `scale_method` controls how the function should scale the output to avoid overflow. If no scaling is selected by setting `scale_method` to zero, then the input signal should be sufficiently conditioned to avoid overflow. The `block_exponent` argument will be set to zero.

The function will perform static scaling if `scale_method` is set to 1. For static scaling, the function will scale intermediate results to prevent overflow. The final output will be scaled by $1/n$, and `block_exponent` will be set to $\log_2(n)$.

If `scale_method` is set to 2, then the function will select dynamic scaling. Under dynamic scaling, the function will inspect the intermediate results and will only scale to avoid overflow. Dynamic scaling therefore minimizes loss of precision but at the cost of slightly reduced performance. The `block_exponent` will be set to a value between 0 (which indicates that no scaling was performed) and $\log_2(n)$ (if static scaling was performed).

Algorithm

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk}$$

The implementation uses core FFT functions implemented as direct radix-4 algorithm. To get the inverse effect, it 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

NxN 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. The function does not impose any special memory alignment requirements on the arrays. However, benefits in run-time performance will be realized if the output array is allocated on an address boundary that is multiple of twice the FFT size. If the input data can be overwritten, then optimum memory usage can be achieved by 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 `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 1.

The arguments `block_exponent` and `scale_method` have been added for future expansion. However, the current version of the function ignores the argument and always scales the output by $n*n$; this is equivalent to static scaling. The function will also set `block_exponent` to $\log_2(n)$.

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 $i=\{0,1,\dots,n-1\}$, $j=\{0,1,2,\dots,n-1\}$

Domain

Input sequence length n must be a power of two and at least 16.

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 */
```

The function uses the following structure to maintain the state of the filter.

```
typedef struct
{
    float *c;            /* coefficients                */
    float *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)
```

DSP Run-Time Library Reference

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 B2, B1, B0, A2, A1. 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

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>
```

```
fract16 mean_fr16(a,n)
```

```
const fract16 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 return the result.

Algorithm

$$c = \frac{1}{n} * (\sum_{i=0}^{n-1} a_i)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ 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

The `mu_compress` function takes a vector of linear 14-bit signed speech samples and performs μ-law compression according to ITU recommendation G.711. Each sample is compressed to 8 bits and is returned in the vector pointed to by `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

The `mu_expand` function inputs a vector of 8-bit compressed speech samples and expands them according to ITU recommendation G.711. Each input value is expanded to a linear 14-bit signed sample in accordance with the μ -law definition and is returned in the vector pointed to `out`.

Algorithm

```
C(k)= mu_law expansion of A(k)

for k=0 to n-1
```

Domain

Content of input array: 0 to 255

DSP Run-Time Library Reference

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 $+3.4 \times 10^{38}$ for `normf ( )`

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

$$\text{Re}(c) = r \cdot \cos(q)$$

$$\text{Im}(c) = r \cdot \sin(q)$$

where q is the phase, and r is the magnitude

Domain

phase = [-9099 ... 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 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. The function does not impose any special memory alignment requirements on the arrays. However, benefits in run-time performance will be realized if the output array is allocated on an address boundary that is multiple of twice the FFT size. If the input data can be overwritten, then optimum memory usage can be achieved by specifying the input array as either the output array or as the temporary array provided that the memory size of the input array is at least $2*n$. Specifying the input array as the temporary array will also result in increased run-time performance.

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 argument `scale_method` controls how the function should scale the output to avoid overflow. If no scaling is selected by setting `scale_method` to zero, then the input signal should be sufficiently conditioned to avoid overflow. The `block_exponent` argument will be set to zero.

The function will perform static scaling if `scale_method` is set to 1. For static scaling, the function will scale intermediate results to prevent overflow. The final output will be scaled by $1/n$, and `block_exponent` will be set to $\log_2(n)$.

If `scale_method` is set to 2, then the function will select dynamic scaling. Under dynamic scaling, the function will inspect the intermediate results and will only scale to avoid overflow. Dynamic scaling therefore minimizes loss of precision but at the possible cost of slightly reduced performance. The `block_exponent` argument will be set to a value between 0 (which indicates that no scaling was performed) and $\log_2(n)$ (as if static scaling was performed).

Algorithm

See [on page 3-31](#).

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 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. The function does not impose any special memory alignment requirements on the arrays. However, benefits in run-time performance will be realized if the output array is allocated on an address boundary that is multiple of twice the FFT size. If the input data can be overwritten, then optimum memory usage can be achieved by specifying the input array as either the output array or as the temporary array provided that the memory size of the input array is at least $2*n$. Specifying the input array as the temporary array will also result in increased run-time performance.

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 argument `scale_method` controls how the function should scale the output to avoid overflow. If no scaling is selected by setting `scale_method` to zero, then the input signal should be sufficiently conditioned to avoid overflow. The `block_exponent` argument will be set to zero.

The function will perform static scaling if `scale_method` is set to 1. For static scaling, the function will scale intermediate results to prevent overflow. The final output will be scaled by $1/n$, and `block_exponent` will be set to $\log_2(n)$.

If `scale_method` is set to 2, then the function will select dynamic scaling. Under dynamic scaling, the function will inspect the intermediate results and will only scale to avoid overflow. Dynamic scaling therefore minimizes loss of precision but at the possible cost of slightly reduced performance. The `block_exponent` argument will be set to a value between 0 (which indicates that no scaling was performed) and $\log_2(n)$ (as if static scaling was performed).

Algorithm

See [on page 3-31](#).

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

NxN 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 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 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. The function does not impose any special memory alignment requirements on the arrays. However, benefits in run-time performance will be realized if the output array is allocated on an address boundary that is multiple of twice the FFT size.

The twiddle table is passed in the argument `w`, which must contain at least `n/2` 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.

The arguments `block_exponent` and `scale_method` have been added for future expansion. However, the current version of the function ignores the argument and always scales the output by $n*n$; this is equivalent to static scaling. The function will also set `block_exponent` to $\log_2(n)$.

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.

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

-3.4×10^{38} to $+3.4 \times 10^{38}$	for <code>rmsf()</code>
-1.0 to $+1.0$	for <code>rms_fr16()</code>

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 to +3.4 x 10³⁸] for rsqrtf()

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 at least $\frac{1}{2}$ 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 $k = \{0, 1, 2, \dots, n/2 - 1\}$

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 at least $\frac{3}{4}n$, 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.

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 $n/2$, 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:

$$twid_re(k) = \cos\left(\frac{2\pi}{n}k\right)$$

$$twid_im(k) = -\sin\left(\frac{2\pi}{n}k\right)$$

where $k = \{0, 1, 2, \dots, n/2 - 1\}$

Domain

The FFT length n must be a power of two and at least 16.

DSP Run-Time Library Reference

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

-3.4×10^{38} to $+3.4 \times 10^{38}$	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 */
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()

A COMPILER LEGACY SUPPORT

The VisualDSP++ environment and tools provide several types of support for legacy code that was developed with previous releases of the development tools. For more information on legacy code support, see the *VisualDSP++ 3.0 Linker and Utilities Manual for ADSP-218x and ADSP-219x DSPs* and *VisualDSP++ 3.0 C/C++ Assembler and Preprocessor Manual for ADSP-218x and ADSP-219x DSPs*.

Tools Differences

VisualDSP++ 3.0 includes an updated C/C++ compiler, linker, and debugger, and a binary file format, ELF. Due to use of the VisualDSP++ Integrated Development and Debugging Environment (IDDE) and other enhancements, VisualDSP++ 3.0 has significant differences from Release 6.1 that you will need to be aware of. In some cases you will need to modify your sources to use the new tools.

Of the new features and enhancements, the following have the most impact on your existing projects:

- Some tools switches have changed. If you use any of the modified or obsolete switches, you must revise your command line scripts or batch files in order to rebuild your project.
- The code generation tools no longer support AEXE-format DSP executables (.EXE). They now generate ELF-format DSP executables (.DXX), and the debugger requires DSP executables to be in the ELF/DWARF-2 format. As a result, AEXE-formatted files must be

Tools Differences

recompiled or reassembled in order to be debugged under VisualDSP++ 3.0. An ELF/DWARF-to-AEXE conversion utility is available in VisualDSP++ 3.0 that will perform back-conversion. An AEXE-to-ELF conversion utility performs forward conversion.

- Some assembly instructions and directives have changed from the VisualDSP 6.1 syntax, but a `-legacy` assembler switch has been provided to assemble files in the old syntax. You may need to review diagnostic messages and revise your source code in order to reassemble your source. Legacy syntax and the new syntax under VisualDSP++ 3.0 cannot be used together in the same source file. They can be mixed together within the same project, as long as they are assembled in different source files.
- Some C compiler extensions to the ISO/ANSI standard have changed. If you use any of the modified or removed extensions, you must revise your code in order to rebuild your project.
- The run-time model has changed. If you call a VisualDSP Release 6.1 assembly language subroutine from your C/C++ program, you must revise the assembly code to comply with the new rules for the C/C++ run-time environment.
- The Architecture File (`.ACH`) is no longer supported. If you re-link using your Release 6.1 object files or object libraries, you must create a Linker Description File for each object or object library before using the new Linker.

The remainder of this section describes these and other known differences between VisualDSP Releases 6.1 and VisualDSP++ 3.0. It also provides assistance when possible for making these required changes.

C/C++ Compiler and Run-time Library

The new `cc219x` compiler provided in VisualDSP++ 3.0 does not support some switches and extensions that were available in the `g21` compiler. As a result, the compiler supports a set of new rules for the run-time environment. This section lists the extensions and switches that have been removed, replaced, or whose function works differently than in Release 6.1.

For further details about the `cc219x` compiler, see Chapter 1, “[Compiler](#)”.

Segment Placement Support Keyword Changed to Section

The `segment()` placement keyword has changed to `section()`. The `section()` construct now precedes the variable declaration, and its argument is a string. For example:

```
section("my_sec") int myvar;
```

For more information about the `section()` construct, see “[Placement Support Keyword \(section\)](#)” on page 1-74.

G21 Compatibility Call

The `cc219x` compiler provides a special G21 compatibility call that enables use of existing libraries with the new compiler. The extern `OldAsmCall` declaration can be added to the prototype(s) of the functions developed under Release 6.1. Your programs will be faster, smaller, and more reliable after the C code is upgraded to use the new compiler.



This convention is similar to the C++/C linkage specification.

Support for G21-Based Options And Extensions

The `cc219x` compiler supports most of the switches and extensions of the previous GNU-based compiler release. For a list of absolute or modified options, see [“Compiler Switch Modifications” on page A-5](#).

ANSI C Extensions

The following extensions are no longer supported, or their functions have been modified:

- `typedef` — This extension was used to define the type of an expression.
- Complex types: `complex`, `creal`, `cimag`, and `conj` — These extensions were used to define complex numbers. Although you cannot write complex number literals, you can have a complex type defined with real and imaginary components. These types need to be managed by the programmer. Support for complex types using such an approach is used in the `libdsp` definitions and use of various complex types.
- Compound statements within expressions — This extension was used to declare variables within an expression. You can achieve these results using `inline` functions.
- Iterator types: `iter` and `sum` — These extensions created loop expressions that were used as a shorthand for working with arrays.
- Assigning variables to specific registers: `asm` — This extension was used to declare a variable and specify a machine register in which to store it.

If you use any of these extensions in your C source code, revise that source.

Compiler Switch Modifications

The switches listed in [Table A-1](#) have been removed or their actions have been modified. If you use any of these switches to compile your C/C++ code, remove or replace the switch before recompiling the code with the new cc219x compiler.

Table A-1. C/C++ Compiler — Obsolete and Replaced Switches

Release 6.1 Switch	Operation under Release 6.1	Change for VisualDSP++ 3.0
-a	Specify Architecture File.	Replaced with -T <Linker Description File> in linker command line. Subsumed in VisualDSP++ build process.
-dD, -dM, and -dN	Output the results of preprocessing and/or list of #defines.	Removed.
-deps	Instruct driver to rebuild only components of a target that have changed.	Removed.
-fno-<high med low>	Disable specified optimization level.	Removed. Optimization controlled with -O switches.
-fcond-mismatch	Allow conditional expression mismatch.	Removed.
-finline-functions	Force function inlining.	Removed.
-fkeep-inline-funtions	Force keeping of inlined functions.	Removed.
-fno-asm	Don't recognize asm as a keyword.	Replaced with -no-extra-keywords (see on page 1-32)
-fno-builtin	Don't recognize builtin functions.	Replaced with -no-builtin (see on page 1-32)

Table A-1. C/C++ Compiler — Obsolete and Replaced Switches (Cont'd)

Release 6.1 Switch	Operation under Release 6.1	Change for VisualDSP++ 3.0
-fsigned-bitfields -funsigned-bitfields	Control whether bit field is signed or unsigned.	Removed. Bit field is signed or unsigned based on the sign of the type definition declaring the bitfield.
-fno-signed-bitfields -fno-unsigned-bitfields	Negative form of the -fsigned-bitfield and -funsigned-bitfield.	Removed. Bit field is signed or unsigned based on the sign of the type definition declaring the bitfield.
-fsigned-char -funsigned-char	Specify whether to default to signed or unsigned char type.	Replaced with -signed-char and -unsigned-char.
-fsyntax-only	Check syntax only; no output.	Replaced with -syntax-only (see page 1-41)
-fwritable-strings	Store string constants in the writable data segment.	Removed. String constants are placed in the seg_data1 section. Definition in the.LDF can place this section in RAM or ROM.
-imacros	Process macro file.	Removed.
-MD and -MMD	Output rules for the make utility; used with -E.	Removed.
-mboot-page=	Specify boot page.	Removed. The ADSP-219x processors do not support paging.
-mdmdata= -mpmdata= -mdcode=	Specify target architecture file segments.	Removed. You can control placement of object file segments using the SECTIONS command in the Linker Description File.
-mlistm	Merge C code with assembler-generated code.	Removed.

Table A-1. C/C++ Compiler — Obsolete and Replaced Switches (Cont'd)

Release 6.1 Switch	Operation under Release 6.1	Change for VisualDSP++ 3.0
<code>-mno-doloops</code>	Do not generate loop structures in assembled code.	Removed. The compiler only generates do loop control structures when it is safe to do so and the compiler is generating optimized code (<code>-O</code> , <code>-Os</code>). Interrupt handling routines should save and restore the loop stack if they are to use the same construct to avoid overflowing the loop stacks.
<code>-mno-inits</code>	Do not initialize variables in assembled code.	Removed.
<code>-mpjump</code>	Place the jump table in pm memory.	Removed.
<code>-mreserved=</code>	Instructs the compiler not to use specified registers.	Removed. Replaced with <code>-reserve</code> (see on page 1-39)
<code>-mrom</code>	Make the module a ROM module.	Removed.
<code>-msmall-code</code>	Optimize for size, not for speed.	Removed. Replaced with <code>-Os</code> (see on page 1-35)
<code>-mstatic-spill</code>	Use dm memory when all register are used.	Removed.
<code>-nostdinc</code>	Do not search standard system directories for header files.	Replaced with <code>-no-std-inc</code> (see on page 1-33)
<code>-nostdlib</code>	Do not use standard system libraries and startup files when linking.	Replaced with <code>-no-std-lib</code> (see on page 1-34)

Table A-1. C/C++ Compiler — Obsolete and Replaced Switches (Cont'd)

Release 6.1 Switch	Operation under Release 6.1	Change for VisualDSP++ 3.0
-runhdr	Specify a particular runtime header.	Removed.
-traditional-cpp	Support some preprocessing features.	Removed.

New and Obsolete Warnings

The VisualDSP++ 3.0 compiler includes several new warning switches. These switches control the number and type of messages reported during a given compilation. They are described in [Table A-2 on page A-8](#).

Table A-2. C/C++ Compiler — New Warning Switches

VisualDSP ++ 3.0 Warning Switch	Description
-warn-protos	Produce a warning when a function is called without a full prototype.
-Wdriver-limit <i>number</i>	Set a maximum <i>number</i> of driver errors.
-Werror-limit <i>number</i>	Set a maximum <i>number</i> of compiler errors.
-Wremarks	Indicates that the compiler may issue remarks, which are diagnostic messages even milder than warnings.
-Wterse	Enable terse warnings.
-pedantic	Causes the compiler to generate warnings for any constructs in a C or C++ source file that does not conform to the ANSI standard.
-W<error remark suppress warn> <num> [, num ...]	Overrides the severity of specific compilation diagnostic messages, where <i>num</i> is the number representing the message to override.

The Release 6.1 warning switches that are no longer supported are listed in [Table A-3 on page A-9](#):

Table A-3. C Compiler — Obsolete Warning Switches

-Wall	-Wformat	-Wswitch
-Wchar-subscripts	-Wimplicit	-Wtrigraphs
-Wcomment	-Wparentheses	-Wuninitialized
-Werror	-Wreturn-type	-Wunused

See [“Compiler Command-Line Switches” on page 1-12](#) for further information on the compiler’s switch set.

Run-Time Model

The cc219x compiler in VisualDSP++ 3.0 produces code that is not fully compatible with the Release 6.1 run-time model. VisualDSP++ 3.0 has significant changes in the registers and stack usage. These changes are especially important if you call an assembly language subroutine from a C/C++ program, or a C/C++ function from an assembly language program. For more information about the VisualDSP++ 3.0 run-time model, see [“C/C++ Run-Time Model and Environment” on page 1-109](#).

C/C++ Run-Time Library

This release includes a set of documented ANSI standard routines that you can call from your C/C++ programs. Many of these routines have been modified to provide better support for the improved performance and code compilation of cc219x.

[Table A-4](#) lists the C/C++ run-time libraries, that have been modified for VisualDSP++ 3.0.

Table A-4. C/C++ Compiler — Modified C/C++ Run-Time Library Functions

Header File	Purpose	Change for VisualDSP++ 2.0
math.h	Mathematics	Added functions: acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, frexp, ldexp, log, log10, modf, pow, sin, sinh, sqrt, tan, tanh
signal.h	Signal Handling	Improved functions: clear_interrupt, interrupt, raise, signal
stdio.h	Input/Output	Added function: perror
stdlib.h	Standard Library	Removed functions: isascii, toascii
ctype.h	Character Handling	Added function: strtod

The cc219x compiler release now includes a complete documented DSP library (libdsp.dlb) and associated include files, providing efficient standard functions required by DSP application designers. For details, refer to Chapter 3, “[DSP Run-Time Library](#)”.



Future cc219x compiler releases may include additional library functions. For complete information on the library contents, see Chapter 2, “[C/C++ Run-Time Library](#)”.

I INDEX

Symbols

- +tdebug-types compiler switch [1-25](#)
- .IDL file [1-107](#)
- @ filename (command file) compiler switch [1-22](#)
- __GROUPNAME__ macro [1-41](#)
- __HOSTNAME__ macro [1-41](#)
- __MACHINE__ macro [1-41](#)
- __REALNAME__ macro [1-41](#)
- __SYSTEM__ macro [1-41](#)
- __USERNAME__ macro [1-41](#)
- μ-law compression (mu_compress function) [3-82](#)
- μ-law expansion (mu_expand function) [3-83](#)

Numerics

- 2192-12 (compile for ADSP-2192-12) compiler switch [1-22](#)
- 219x (compile for ADSP-219x family) compiler switch [1-23](#)
- 2-D convolution (conv2d function) [3-44](#)
- 2-D convolution (conv2d3x3 function) [3-45](#)

A

- A (assert) compiler switch [1-23](#)
- a_compress (A-law compression) function [3-22](#)
- a_expand (A-law expansion) function [3-23](#)
- abend (*see* abort function)
- abort (abnormal program end) function [2-25](#)
- Abridged C++ Library
 - overview [2-2](#)
 - support [2-14](#)
- abs (absolute value) function [2-26](#)
- acos (arc cosine) function [2-27](#)
- acos_fr16() function [2-27](#)
- aggregate constructor expression
 - support [1-81](#)
- align num pragma [1-100](#)
- allocate memory (*see* calloc, free, malloc, realloc functions)
- alphanumeric character test (*see* isalnum function)
- altregisters pragma [1-101](#)
- alttok (alternative tokens) C++ mode compiler switch [1-23](#)

INDEX

- analog (Analog C compilation)
 - C/C++ mode selection switch [1-21](#)
- ANSI standard [2-1](#)
 - compiler [1-27](#)
 - extensions to [1-21](#)
- arg (get phase of complex number)
 - function [3-24](#)
- arithmetic, saturated [1-84](#)
- array length [1-77](#)
- array search, binary (*see* [bsearch](#) function)
- ASCII string (*see* [atof](#), [atoi](#), [atol](#) functions)
- asin (arc sine) function [2-28](#)
- asin_fr16() function [2-28](#)
- asm compiler keyword [1-55](#)
- asm compiler keyword (*see also* [Inline assembly language support keyword \(asm\)](#))
- assembly
 - construct
 - operand [1-62](#)
 - reordering and optimization [1-66](#)
 - template [1-59](#)
 - with input and output operands [1-67](#)
 - constructs
 - with multiple instructions [1-65](#)
 - subroutines [1-123](#)
- assignments
 - memory [1-71](#)
- atan (arc tangent) function [2-29](#)

- atan_fr16() function [2-29](#)
- atan2 (arc tangent of quotient)
 - function [2-30](#)
- atan2, atan2f (arc tangent division)
 - functions [2-31](#)
- atan2_fr16() function [2-30](#)
- atexit (select exit function) function [2-31](#)
- atof (convert string to double)
 - function [2-32](#)
- atoi (convert string to integer)
 - function [2-33](#)
- atol (convert string to long integer)
 - function [2-34](#)
- autocoh (autocoherence) function [3-25](#)
- autocorr (autocorrelation) function [3-26](#)
- automatic variables [1-69](#)

B

- binary array search (*see* [bsearch](#) function)
- bool (accept boolean) C++ mode compiler switch [1-46](#)
- bool (*see* [Boolean type support keywords \(bool, true, false\)](#))
- boolean type keywords [1-75](#)
- Boolean type support keywords ([bool](#), [true](#), [false](#)) [1-55](#), [1-75](#)
- bsearch (binary search in sorted array) function [2-35](#)
- build-lib (build library) compiler switch [1-24](#)

INDEX

built-in functions 1-85, 1-86

C

-C (comments) compiler switch
1-24

-c (compile only) compiler switch
1-24

C language extensions 1-55

aggregate assignments 1-81

asm keyword 1-58

bool keyword 1-75

C++ style comments 1-56

compound statements 1-78

false keyword 1-55

indexed initializers 1-79

inline keyword 1-57

non-constant initializers 1-78

segment keyword 1-55

true keyword 1-55

C run-time library

header files 2-8–2-13

reference 2-21–2-150

C run-time model 1-109

C type functions

isalnum 2-63

isctrl 2-65

isgraph 2-67

islower 2-68, 2-70

isprint 2-73

ispunct 2-74

isspace 2-75

isupper 2-76

isxdigit 2-77

tolower 2-145

toupper 2-146

-c++ (C++ mode) compiler switch
1-21

C++ header files for library facilities
2-17–??

C++ mode compiler 1-49

C++ programming examples 1-129

complex support 1-130

fract support 1-129

C++ run-time library

and linker 2-5

C/C++ run-time

environment

(See also mixed C/C++/assembly programming)

C/C++ run-time library

guide 2-3–2-20

cabs (complex absolute value)

function 3-27

cadd (complex addition) function
3-28

callee preserved registers 1-117

caller save registers 1-117

calling assembly language

subroutine 1-123

calling library functions 2-3

DSP 3-2

calloc (allocate and initialize

memory) function 2-37

cc219x 1-103

cdiv (complex division) function
3-29

ceil (ceiling) function 2-38

INDEX

- cexp (complex exponential)
function 3-30
- cfft (N point complex input FFT)
function 3-31
- cfft2d (NxN point 2-D complex
input FFT) 3-35
- cfft4d (N point complex input
FFT) 3-33
- cfir (complex FIR filter) function
3-37
- cfir (finite impulse response filter)
function 3-37
- character string search (*see* strchr
function)
- character string search, recursive (*see*
strchr function)
- circular buffer length registers 1-118
- clear_interrupt (clear a pending
signal) function 2-39
- clip (clip) function 3-39
- cmlt (complex multiply) function
3-40
- command-line interface 1-6–1-53
- compiler
 - C extensions 1-55
 - legacy support ??–A-10
- compiler optimization
 - disabling 1-34
- complex conjugate (conj function)
3-41
- complex multiply (cmlt function)
3-40
- complex number support 1-130
- complex subtraction (csub function)
3-51
- compound statements
 - within expressions 1-78
- compound statements as macros
1-105
- conj (complex conjugate) function
3-41
- const keyword 1-21
- construct from polar coordinates
(polar function) 3-85
- construct operand
 - assembly 1-62
- constructs
 - reordering and optimization 1-66
 - with input and output operands
1-67
 - with multiple instructions 1-65
- control character test (*see* iscntrl
function)
- conv2d (2-D convolution) function
3-44
- conv2d3x3 (2-D convolution)
function 3-45
- convert, characters (*see* tolower,
toupper functions)
- convert, strings (*see* atof, atoi, atol,
strtok, strtol, strtoul, functions)
- convolve (convolution) function
3-42
- convolve transformations 3-6
- copysign (copysign) function 3-46
- cos (cosine) function 2-41
- cos_fr16() function 2-41

INDEX

cosh (hyperbolic cosine) function
2-42
cot (cotangent) function 3-47
count zero crossing (zero_cross
function) 3-101
countones (count one bits in word)
function 3-48
crosscoh (cross-coherence) function
3-49
crosscorr (cross-correlation)
function 3-50
csub (complex subtraction) function
3-51

D

-D (define macro) compiler switch
1-25, 1-42
DAG1 registers 1-120
DAG2 registers 1-121
data register 1-119
deallocate memory (*see* free
function)
debugging information 1-40
dedicated registers 1-117
-default-linkage-asm (assembler
linkage) compiler switch 1-26
-default-linkage-C (C linkage)
compiler switch 1-26
-default-linkage-C++ (C++ linkage)
compiler switch 1-26
disabling compiler optimization
1-34
div (division) function 2-44
div (division, int) function 2-63

division (*see* div, ldiv functions)
dm memory keyword 1-55
-dollar (dollar format) compiler
switch 1-26
double representation 2-127
double word data types 1-53
-dry (verbose dry-run) compiler
switch 1-26
-dryrun (terse -dry-run) compiler
switch 1-26
DSP
run-time library format 3-21
DSP functions 3-21–3-101
DSP header files 3-4–3-20
DSP run-time library
functions
linking 3-3
guide 3-2
source code 3-3
dual memory support keywords
(pm dm) 1-55, 1-69
illegal 1-73

E

-E (stop after preprocessing)
compiler switch 1-26
-EE (run after preprocessing)
compiler switch 1-27
elfar (archive library) 1-2
elfar)archive library) 1-24
Embedded C++ Library
header files 2-14–2-17
Embedded Standard Template
Library 2-2

INDEX

- header files [2-18–2-20](#)
- enable_interrupts (enable interrupts) function [2-45](#)
- end (*see* atexit, exit functions)
- ETSI run-time library [2-6](#)
- ETSI support [1-91](#)
- exit (normal program termination) function [2-46](#)
- exit (program termination) function [2-46](#)
- exp (exponential) function [2-47](#)
- explicit (explicit specifier) C++ mode compiler switch [1-46](#)
- extensions
 - using compiler language [1-3](#)
- external memory
 - accessing [1-90](#)
- external_memory_read function [2-48](#)
- external_memory_write function [2-50](#)
- extra-keywords (enable short-form keywords) compiler switch [1-27](#)

F

- fabs (float absolute value) function [2-52](#)
- false (*see* Boolean type support keywords (bool, true, false))
- far jump return (*see* longjmp, setjmp functions)
- Fast Fourier Transforms [3-6](#)
- FFT function versions [3-6](#)
- file

- extension [1-7, 1-9](#)
- names [1-22](#)
- searches [1-9](#)
- file searching
 - <filename> [1-107](#)
- filter.h header file [3-6](#)
- filters
 - in digital signal processing [3-6](#)
- finite impulse response filter (cfir function) [3-37](#)
- finite impulse response filter (fir function) [3-52](#)
- fir (finite impulse response filter) function [3-52](#)
- fir_decima (FIR decimation filter) function [3-54](#)
- fir_interp (FIR interpolation filter) function [3-56](#)
- flags (command line input) compiler switch [1-27](#)
- float representation [2-129](#)
- floor (floor) function [2-53](#)
- fmod (floating-point modulus) function [2-54](#)
- force definitions (force vtable definitions) C++ mode compiler switch [1-47](#)
- fract [1-129](#)
 - data type (C++ mode) [1-82](#)
- fract type support [1-129](#)
- fractional
 - arithmetic operations [1-83](#)
 - literal format [1-82](#)
 - mixed-mode operations [1-84](#)

INDEX

- saturated arithmetic [1-84](#)
- type conversions [1-83](#)
- fractional type support [1-82](#)
- free (deallocate memory) function [2-55](#)
- free (deallocate memory) functions [2-63](#)
- frexp (separate fraction and exponent) function [2-56](#)
- full-version (display versions) compiler switch [1-27](#)
- function arguments [1-73](#) and memory keywords [1-73](#)
- functions
 - built-in [1-86](#)
 - C run-time library [2-24–2-150](#)
 - DSP [3-21–3-101](#)
 - primitive I/O [2-12](#)
 - program control
 - calloc [2-37](#)
 - free [2-63](#)
 - malloc [2-85](#)
 - realloc [2-102](#)

G

- g (generate debug information) compiler switch [1-28](#)
- gen_bartlett (generate bartlett window) function [3-58](#)
- gen_blackman (generate blackman window) function [3-60](#)
- gen_gaussian (generate gaussian window) function [3-61](#)

- gen_hamming (generate hamming window) function [3-62](#)
- gen_hanning (generate hanning window) function [3-63](#)
- gen_harris (gen_harris window) function [3-64](#)
- gen_kaiser (generate kaiser window) function [3-65](#)
- gen_rectangular (generate rectangular window) function [3-66](#)
- gen_triangle (generate triangle window) function [3-67](#)
- gen_vohann (generate von hann window) function [3-69](#)
- get phase of a complex number (arg function) [3-24](#)
- graphical character test (*see* isgraph function)

H

- H (list headers) compiler switch [1-28](#)
- hardware call stack [1-33](#)
- header files [1-105, 3-4](#)
 - C run-time library [2-8–2-13](#)
 - assert.h [2-9](#)
 - ctype.h [2-9](#)
 - def2191.h [2-10](#)
 - def2192-12.h [2-10](#)
 - def219x.h [2-10](#)
 - errno.h [2-10](#)
 - Float.h [2-10](#)
 - limits.h [2-11](#)

INDEX

- locale.h [2-11](#)
- Math.h [2-11](#)
- setjmp.h [2-12](#)
- signal.h [2-12](#)
- stdarg.h [2-12](#)
- stddef.h [2-12](#)
- stdio.h [2-12](#)
- stdlib.h [2-13](#)
- string.h [2-13](#)
- sysreg.h [2-13](#)
- C++ access to C facilities
 - cassert [2-17](#)
 - cctype [2-17](#)
 - cerrno [2-18](#)
 - cfloat [2-18](#)
 - climits [2-18](#)
 - locale [2-18](#)
 - cmath [2-18](#)
 - csetjmp [2-18](#)
 - csignal [2-18](#)
 - cstdarg [2-18](#)
 - cstddef [2-18](#)
 - cstdio [2-18](#)
 - cstdlib [2-18](#)
 - cstring [2-18](#)
- DSP [3-4–3-20](#)
 - complex.h [3-4–3-6](#)
 - filter.h [3-6–??](#)
 - math.h [3-10–3-12](#)
 - matrix.h [3-12–3-15](#)
 - stats.h [3-15–3-16](#)
 - vector.h [3-16–3-19](#)
 - window.h [3-20](#)
- Embedded C++ Library
 - complex [2-14](#)
 - exception [2-14](#)
 - fract [2-15](#)
 - fstream [2-15](#)
 - iomanip [2-15](#)
 - ios [2-15](#)
 - iosfwd [2-15](#)
 - iostream [2-15](#)
 - istream [2-16](#)
 - new [2-16](#)
 - ostream [2-16](#)
 - sstream [2-16](#)
 - stdexcept [2-16](#)
 - streambuf [2-16](#)
 - string [2-17](#)
 - strstream [2-17](#)
- Embedded Standard Template
 - Library [2-18–2-20](#)
 - algorithm [2-18](#)
 - deque [2-18](#)
 - fstreams.h [2-20](#)
 - functional [2-19](#)
 - hash_map [2-19](#)
 - hash_set [2-19](#)
 - iomanip.h [2-20](#)
 - iostream.h [2-20](#)
 - iterator [2-19](#)
 - list [2-19](#)
 - map [2-19](#)
 - memory [2-19](#)
 - new.h [2-20](#)
 - numeric [2-19](#)
 - queue [2-19](#)
 - set [2-20](#)

INDEX

- stack [2-20](#)
- utility [2-20](#)
- vector [2-20](#)
- system [1-105](#)
- user [1-105](#)
- help (command line help) compiler switch [1-28](#)
- hexadecimal digit test (*see* `isxdigit` function)
- HH (list headers and compile) compiler switch [1-28](#)
- histogram [3-70](#)

I

- I (start include directory) compiler switch [1-29](#), [1-33](#)
- I/O space addresses [1-88](#)
- ifft (N point inverse FFT) function [3-71](#)
- ifft2d (NxN point 2-D inverse input FFT) function [3-75](#)
- ifft2rad4 (N point inverse FFT) function [3-73](#)
- iir (infinite impulse response filter) function [3-77](#)
- iir_init macro [3-77](#)
- include (include file) compiler switch [1-29](#)
- include directives [1-107](#)
- indexed initializer support [1-78](#)
- inline assembly language support keyword (asm) [1-58](#)
- construct
 - I/O operands [1-67](#)

- optimization [1-66](#)
- template [1-59](#)
- template operands [1-63](#)
- with multiple instructions [1-66](#)
- construct template [1-59](#)
- macros containing asm [1-68](#)
- inline function support keyword (inline) [1-21](#), [1-55](#), [1-57](#)
- instantall (instantiate all templates) compiler switch [1-47](#)
- instantiation of templates [1-47](#)
- instantlocal (instantiate used with internal linkage) compiler switch [1-47](#)
- instantused (instantiate used) compiler switch [1-47](#)
- interfacing C/C++ and assembly (See mixed C/C++/assembly programming)
- interrupt (define interrupt handling) function [2-57](#)
- interrupt pragma [1-100](#), [1-101](#)
- interruptf function [2-57](#)
- Interrupts (*see* `clear_interrupt`, `interruptf`, `interrupts`, `signal`, `raise` functions)
- io_space_read (read I/O space) function [2-61](#)
- io_space_write (write I/O space) function [2-62](#)
- isalnum (detect alphanumeric character) function [2-63](#)
- isalpha (detect alphabetic character) function [2-64](#)

INDEX

isctrl (detect control character)
 function [2-65](#)
isdigit (detect decimal digit)
 function [2-66](#)
isgraph (detect printable character)
 function [2-67](#)
isinf (test for infinity) function [2-68](#)
islower (detect lowercase character)
 function [2-70](#)
isnan (test for NAN) function [2-71](#)
isprint (detect printable character)
 function [2-73](#)
ispunct (detect punctuation
 character) function [2-74](#)
isspace (detect whitespace character)
 function [2-75](#)
isupper (detect uppercase character)
 function [2-76](#)
isxdigit (detect hexadecimal digit)
 function [2-77](#)

J

-J compiler switch [1-30](#)

K

keywords
 dual memory support
 illegal [1-73](#)
 memory [1-71](#), [1-73](#)
 memory and macros [1-74](#)
keywords (compiler) (*see*
 VisualDSP++ compiler C/C++
 language extensions)

L

-L (library search directory)
 compiler switch [1-30](#)
-l (link library) compiler switch
 [1-30](#), [1-34](#)
labs (long integer absolute value)
 function [2-78](#)
Language extensions (compiler) (*see*
 VisualDSP++ compiler C/C++
 language extensions)
ldexp (multiple by power of 2)
 function [2-79](#)
ldiv (division, long) function [2-80](#)
legacy support
 compiler ??-[A-10](#)
libetsi.dlb [1-95](#)
library
 format [3-21](#)
 format for DSP run-time [3-21](#)
 functions, listed [2-21](#)
 header files, working with [2-8](#)
 source code, working with [2-7](#)
linkage pragmas [1-101](#)
linkage_name pragma [1-101](#), [1-102](#)
linking DSP run-time library
 functions [3-3](#)
log (natural logarithm) function
 [2-81](#)
log10 (base 10 logarithm) function
 [2-82](#)
Long jump (*see* longjmp, setjmp
 functions)
longjmp (second return from
 setjmp) function [2-83](#)

INDEX

lower case (*see* `islower`, `tolower` functions)

M

-M (generate make rules only)
 compiler switch [1-30](#)

macros

 and `asm()` C program constructs
 [1-68](#)

 compound statements as [1-105](#)

 EDOM [2-11](#)

 ERANGE [2-11](#)

 HUGE_VAL [2-11](#)

 predefined [1-103](#)

`malloc` (allocate memory) function
 [2-85](#)

-map (generate a memory map)
 compiler switch [1-31](#)

mathematical functions

 floating point [2-11](#)

`max` (maximum) function [3-79](#)

`mean` (mean) function [3-80](#)

`mean` (mean, array) function [2-103](#)

`memchr` (find first occurrence of
 character) function [2-86](#)

`memcmp` (compare objects)
 function [2-87](#)

`memcpy` (copy characters from one
 object to another) function
 [2-88](#)

`memcpy_to_shared` function [2-90](#)

`memmove` (move characters from
 one object to another) function
 [2-91](#)

memory

 assignments [1-71](#)

memory (*see* `calloc`, `free`, `malloc`,
 `memcpy`, `memcpy`, `memset`,
 `memmove`, `memchar`, `realloc`
 functions)

memory allocation [2-102](#)

memory keywords [1-71](#)

 and function arguments [1-73](#)

 and function declarations /

 pointers [1-72](#)

 and macros [1-74](#)

`memset` (set range of memory to a
 character) function [2-92](#)

`min` (minimum) function [3-81](#)

mixed C/assembly

 programming

`asm()` constructs [1-59](#)

mixed C/assembly naming

 conventions [1-127](#)

mixed C/assembly programming

`asm()` constructs [1-58](#), [1-59](#), [1-63](#),
 [1-66](#), [1-67](#), [1-68](#)

mixed C/C++/assembly

 programming [1-109](#)

mixed C/C++/assembly reference
 [1-126](#)

-MM (generate make rules and
 compile) compiler switch [1-31](#)

`mode_change` (change selected
 system modes) function [2-93](#)

`modf` (separate integral and
 fractional parts) function [2-95](#)

INDEX

move memory range (*see* memmove function)

-MQ compiler switch [1-31](#)

MSTAT (mode status) register
[1-118](#)

-Mt filename (output make rule)
compiler switch [1-31](#)

mu_compress (μ -law compression)
function [3-82](#)

mu_expand (μ -law expansion)
function [3-83](#)

multidimensional arrays [1-77](#)

multi-line asm() C program
constructs [1-66](#)

multiple instructions
constructs with [1-65](#)

N

N point complex input FFT (cfft
function) [3-31](#)

N point complex input FFT
(cffttrad4 function) [3-33](#)

N point inverse FFT (ifft function)
[3-71](#)

N point inverse FFT (iffttrad4
function) [3-73](#)

N point real input FFT (rfft
function) [3-86](#)

N point real input FFT (rffttrad4
function) [3-88](#)

-namespace (enable namespaces)
C++ mode compiler switch
[1-48](#)

-newforinit (new ‘for’ initialization)
C++ mode compiler switch
[1-48](#)

-newvec (new vector) C++ mode
compiler switch [1-48](#)

-no_hardware_pc_stack compiler
switch [1-33](#)

-no-alttok (disable tokens) compiler
common switch [1-31](#)

-no-bool (disable boolean) C++
mode compiler switch [1-48](#)

-no-builtin (no builtin functions)
compiler switch [1-32](#)

-no-char (disable wide char type)
C++ mode compiler switch
[1-49](#)

-no-def (disable definitions)
compiler switch [1-32](#)

-no-demangle (disable demangler)
C++ mode compiler switch
[1-48](#)

-no-dir-warnings (disable directory
warning) compiler switch [1-32](#)

-no-dollar (disable dollar format)
C++ mode compiler switch
[1-32](#)

-no-explicit (disable explicit
specifier) C++ mode compiler
switch [1-49](#)

-no-extra-keywords (disable
short-form keywords) compiler
switch [1-32](#)

-no-inline (disable inline keyword)
compiler switch [1-33](#)

INDEX

- no-namespace (disable namespaces) C++ mode compiler switch [1-49](#)
- non-constant initializer support (compiler) [1-78](#)
- no-newvec (disallow a new vector) C++ mode compiler switch [1-49](#)
- no-restrict (no restrict support) C++ mode compiler switch [1-49](#)
- norm (normalization) [3-84](#)
- no-std-def (disable standard definitions) compiler switch [1-33](#)
- no-std-inc (disable standard include search) compiler switch [1-33](#)
- no-std-lib (disable standard library search) compiler switch [1-34](#)
- nothreads (disable thread-safe build) compiler switch [1-34](#)
- notstrict (non-strict compilation) C++ mode compiler switch [1-49](#)
- no-widen-muls (disable widening multiplications) compiler switch [1-34](#)
- NxN point 2-D complex input FFT (cfft2d function) [3-35](#)
- NxN point 2-D inverse input FFT (ifft2d function) [3-75](#)
- NxN point 2-D real input FFT (rfft2d function) [3-90](#)

O

- O (enable optimization) compiler switch [1-35](#)
- o (output) compiler switch [1-35](#)
- optimization pragmas [1-101](#)
- optimizing asm() C program constructs [1-66](#)
- Os (optimize for size) compiler switch [1-35](#)

P

- P (omit line numbers) compiler switch [1-35](#)
- path-install (installation location) compiler switch [1-36](#)
- path-output (non-temporary files location) compiler switch [1-36](#)
- path-temp (temporary files location) compiler switch [1-36](#)
- path-tool (tool location) compiler switch [1-36](#)
- pch (precompiled header) compiler switch [1-37](#)
- pchdir (locate PCHRepository) compiler switch [1-37](#)
- pedantic (ANSI standard warnings) compiler switch [1-37](#)
- pedantic-errors (ANSI C errors) compiler switch [1-37](#)
- placement support keyword (segment) [1-55](#), [1-74](#)
- pm memory keyword [1-55](#), [1-69](#)
- pointer class support keyword (restrict) [1-55](#), [1-75](#)

INDEX

- pointers
 - memory keywords 1-72
 - polar (construct from polar coordinates) function 3-85
 - pow (raise to a power) function 2-96
 - PP (omit line numbers and compile) compiler switch 1-36
 - pplist (preprocessor listing) compiler switch 1-38
 - pragmas 1-99
 - align num 1-100
 - altregisters 1-101
 - interrupt handler 1-100
 - linkage_name 1-102
 - linking 1-101
 - optimization 1-101
 - optimization level 1-101
 - weak_entry 1-102
 - precompiled header 1-37
 - predefined macros 1-103
 - __ADSP2191|95|96|990__ 1-103
 - __ADSP2192_12__ 1-103
 - __ADSP21XX__ 1-103
 - __ANALOG_EXTENSIONS__ 1-103
 - __cplusplus 1-103
 - __DOUBLES_ARE_FLOATS__ 1-104
 - __ECC__ 1-104
 - __EDG__ 1-104
 - __EDG_VERSION__ 1-104
 - __FILE__ 1-104
 - __LINE__ 1-104
 - __NO_BUILTIN 1-104
 - __SIGNED_CHARS__ 1-104
 - __STDC__ 1-104
 - __STDC_VERSION__ 1-104
 - __TIME__ 1-104
 - __LANGUAGE_C 1-104
 - __NO_LONG_LONG 1-104
 - prelinker 1-47
 - preprocessing
 - IDL files 1-107
 - preprocessor macros 1-103
 - primitive I/O functions 2-12
 - printable character test (*see* isprint function)
 - printf (write formatted stream output to standard out) function 2-97
 - proc (target processor) compiler switch 1-38
 - procedure call 1-115
 - program control functions
 - calloc 2-37
 - free 2-63
 - malloc 2-85
 - realloc 2-102
 - punctuation character test (ispunct) function 2-74
- ## Q
- qsort (quicksort) function 2-97
 - qsort function 2-97

INDEX

R

- R- (disable source path) compiler switch [1-39](#)
- R directory (add source directory) compiler switch [1-38](#)
- raise (raise a signal) function [2-99](#)
- rand (random number generator) function [2-101](#)
- random number (*see* rand, srand functions) [2-101](#)
- realloc (change memory allocation) function [2-102](#)
- real-time signals (*see* clear_interrupt, interruptf, interrupts, poll_flag_in, raise, signal functions)
- reciprocal square root (rsqrt) function [3-93](#)
- register classification mode status (MSTAT) [1-118](#)
- registers [1-117](#)
 - DAG1 [1-120](#)
 - DAG2 [1-121](#)
 - data [1-119](#)
 - for asm() constructs [1-63](#)
 - miscellaneous [1-119](#)
- reordering asm() C program constructs [1-66](#)
- reserve (reserve register) compiler switch [1-39](#)
- restrict (support restrict keyword) C++ mode compiler switch [1-50](#)
- return values [1-115](#)

- rfft (N point real input FFT) function [3-86](#)
- rfft2d (NxN point 2-D real input FFT) function [3-90](#)
- rfftrad4 (N point inverse FFT) function [3-88](#)
- rms (root mean square) function [3-92](#)
- rsqrt (reciprocal square root) function [3-92](#)
- run-time environment
 - (See also mixed C/C++/assembly programming)
 - programming (See mixed C/C++/assembly programming)

S

- S (stop after compilation) compiler switch [1-39](#)
- s (strip debugging information) compiler switch [1-40](#)
- saturated arithmetic [1-84](#)
- save-temps (save intermediate files) compiler switch [1-40](#)
- search character string (*see* strchr, strrchr functions)
- search memory, character (*see* memchar function)
- search path
 - for include files [1-29](#)
 - for library files [1-30](#)
- searching for #included files [1-107](#)

INDEX

- segment (*see* placement support keyword (segment))
- set jump (*see* longjmp, setjmp functions)
- setjmp (define runtime label) function 2-103
- setjmp (label for external linkage) function 2-103
- setjmp (long jump) function 2-103
- show (display command line) compiler switch 1-40
- signal (define signal handling) function 2-105
- signal handling 2-12
- signals (*see* clear_interrupt, interruptf, interrupts, poll_flag_in, raise, signal functions)
- signed keyword 1-21
- signed-char (make char signed) compiler switch 1-40
- sin (sine) function 2-108
- sin_fr16() function 2-108
- sinh (hyperbolic sine) function 2-109
- source code
 - DSP run-time library 3-3
 - library 2-7
- sqrt (square root) function 2-110
- srand (Random number seed) function 2-111
- srand (random number seed) function 2-111
- standard conformance 1-4
- stdout 1-26
- Stop (*see* atexit, exit functions)
- strcat (concatenate strings) function 2-112
- strchr (find first occurrence of character in string) function 2-113
- strcmp (compare strings) function 2-114, 2-117
- strcoll (compare strings) function 2-115
- strcpy (copy from one string to another) function 2-116
- strcspn (length of character segment in one string but not the other) function 2-117
- strerror (get string containing error message) function 2-118
- strict (strict compilation) C++ mode compiler switch 1-50
- strictwarn (warn if non-strict) C++ mode compiler switch 1-50
- string conversion (*see* atof, atoi, atol, strtok, strtol, strxfrm functions)
- string functions
 - memchar 2-86
 - memmove 2-91
 - strchr 2-113
 - strcoll 2-115
 - strcspn 2-117
 - strerror 2-118
 - strpbrk 2-123
 - strrchr 2-124, 2-125
 - strspn 2-125

INDEX

- strchr 2-126
- strtok 2-131
- strxfrm 2-137
- strlen (string length) function 2-119
- strncat (concatenate characters from one string to another) function 2-120
- strncmp (compare characters in strings) function 2-121
- strncpy (copy characters from one string to another) function 2-122
- strpbrk (find character match in two strings) function 2-123
- strrchr (find last occurrence of character in string) function 2-124
- strspn (length of segment of characters in both strings) function 2-125
- strstr (find string within string) function 2-126
- strtod (convert portion of string to double) function 2-127
- strtod (convert to double representation) function 2-127
- strtodf (convert to float representation) function 2-129
- strtok (convert string to tokens) function 2-131
- strtol (convert string to long integer) function 2-133
- strtoul (convert string to unsigned long integer) function 2-135
- strxfrm (transform string using LC_COLLATE) function 2-137
- strxfrm function 2-137
- support example 1-129
- suppress (suppress vtable definitions) C++ mode compiler switch 1-50
- switches
 - C++ mode compiler 1-46–1-51
 - bool (accept boolean) 1-46
 - explicit (explicit specifier) 1-46
 - force definitions (force vtable definitions) 1-47
 - instantused (instantiate used) 1-47
 - namespace (enable namespaces) 1-48
 - newforinit (new ‘for’ initialization) 1-48
 - newvec (new vector) 1-48
 - no-bool (disable boolean) 1-48
 - no-char (disable wide char type) 1-49
 - no-demangle (disable demangler) 1-48
 - no-dollar (disable dollar format) 1-32
 - no-explicit (disable explicit specifier) 1-49
 - no-namespace (disable namespaces) 1-49
 - no-newvec (disallow a new vector) 1-49

INDEX

- no-restrict (no restrict support) [1-49](#)
- notstrict (non-strict compilation) [1-49](#)
- restrict (support restrict keyword) [1-50](#)
- strictwarn (warn if non-strict) [1-50](#)
- suppress (suppress vtable definitions) [1-50](#)
- tpautooff (no automatic templates) [1-51](#)
- trdforinit (traditional initialization) [1-51](#)
- typename (support typename) [1-51](#)
- wchar (support wide char type) [1-51](#)
- C/C++ mode selection [1-21](#)
 - analog (Analog C compilation) [1-21](#)
- compiler common [1-22–1-46](#)
 - @ filename (command file) [1-22](#)
 - 2192-12 (compile for ADSP-2192-12) [1-22](#)
 - 219x (compile for ADSP-219x family) [1-23](#)
 - A (assert) compiler switch [1-23](#)
 - build-lib (build library) [1-24](#)
 - C (comments) [1-24](#)
 - c (compile only) [1-24](#)
 - compiler errors [1-44](#)
 - debug-types [1-25](#)
 - default-linkage-asm (assembler linkage) [1-26](#)
 - default-linkage-C (C linkage) [1-26](#)
 - default-linkage-C++ (C++ linkage) [1-26](#)
 - Dmacro (define macro) [1-25](#)
 - dollar (dollar format) [1-26](#)
 - dry (verbose dry-run) [1-26](#)
 - dryrun (terse dry-run) [1-26](#)
 - E (stop after preprocessing) [1-26](#)
 - EE (run after preprocessing) [1-27](#)
 - extra-keywords (enable short-form keywords) [1-27](#)
 - flags (command-line input) [1-27](#)
 - full-version (display versions) [1-27](#)
 - g (generate debug information) [1-28](#)
 - H (list headers) [1-28](#)
 - h[elp] (command-line help) [1-28](#)
 - HH (list headers and compile) [1-28](#)
 - I (start include directory) [1-29](#)
 - I directory (include search directory) [1-29](#)
 - include (include file) [1-29](#)
 - J (old-style preprocessing) [1-30](#)
 - l (link library) [1-30](#)
 - L directory (library search di-

INDEX

- rectory) 1-30
- M (make rules only) 1-30
- map filename (generate a memory map) 1-31
- MM (generate make rules and compile) 1-31
- MQ 1-31
- Mt (output make rules) 1-31
- no_hardware_pc_stack 1-33
- no-builtin (no built-in functions) 1-32
- no-defs (disable defaults) 1-32
- no-dir-warnings 1-32
- no-extra-keywords (disable short-form keywords) 1-32
- no-inline (disable inline keyword) 1-33
- no-std-def (disable standard macro definitions) 1-33
- no-std-inc (disable standard include search) 1-33
- no-std-lib (disable standard library search) 1-34
- nothreads (disable thread-safe build) 1-34
- no-widen-muls (disable widening multiplications) 1-34
- O (enable optimizations) 1-35
- o filename (output file) 1-35
- Os (optimize for size) 1-35
- P (omit line numbers) 1-35
- path (tool location) 1-36
- path-install directory (installation location) 1-36
- path-output directory (non-temporary files location) 1-36
- path-temp directory (temporary files location) 1-36
- pch (precompiled header) 1-37
- pchdir (locate PCHRepository) 1-37
- pedantic (ANSI standard warnings) 1-37
- pedantic-errors (ANSI C errors) 1-37
- PP (omit line numbers and compile) 1-36
- pplist file (preprocessor listing) 1-38
- proc identifier 1-38
- R- (disable source path) 1-39
- R directory (add source directory) 1-38
- reserve (reserve register) 1-39
- S (stop after compilation) 1-39
- s (strip debugging information) 1-40
- save-temps (save intermediate files) 1-40
- show (display command line) 1-40
- signed-char (make char signed) 1-40
- sourcefile (sourcefile parameter) 1-22
- syntax-only (just check syntax) 1-41

INDEX

- syntax-only (system definitions) [1-41](#)
- T (linker description file) [1-41](#)
- threads (enable thread-safe build) [1-42](#)
- time (time the compiler) [1-42](#)
- traditional (traditional C compilation) [1-21](#)
- Umacro (undefine macro) [1-42](#)
- unsigned-char (make char unsigned) [1-42](#)
- v (version and verbose) [1-42](#)
- val-global (add global names) [1-43](#)
- verbose (display command line) [1-43](#)
- version (display version) [1-43](#)
- w (disable all warnings) [1-43](#)
- warn-protos (prototypes warning) [1-45](#)
- Wdriver-limit (maximum process errors) [1-44](#)
- Werror-limit (maximum switches) [1-44](#)
- workaround [1-45](#)
- Wremarks (enable diagnostic warnings) [1-44](#)
- write-files (enable file redirection) [1-45](#)
- write-opts [1-45](#)
- Wterse (enable terse warnings) [1-44](#)
- xref file (cross-reference list) [1-45](#)

- syntax-only (just check syntax) compiler switch [1-41](#)
- sysdef (system definitions) compiler switch [1-41](#)
- sysreg.h header file [1-86](#)
- sysreg_read (read from non-memory-mapped register) function [2-139](#)
- sysreg_write (write to non-memory-mapped register) function [2-141](#)
- system header files [1-105](#)

T

- T (linker description file) compiler switch [1-41](#)
- tan (tangent) function [2-143](#)
- tan_fr16() function [2-143](#)
- tanh (hyperbolic tangent) function [2-144](#)
- template
 - assembly construct [1-59](#)
 - for asm() in C programs [1-59](#)
- template for asm() in C programs [1-59](#)
- Terminate (*see* atexit, exit functions)
- threads (enable thread-safe build) compiler switch [1-42](#)
- time (time the compiler) switch [1-42](#)
- Tokens, string convert (*see* strtok function)
- tolower (convert from uppercase to lowercase) function [2-145](#)

INDEX

toupper (convert from lowercase to uppercase) function [2-146](#)
-tpautooff (no automatic templates) C++ mode compiler switch [1-51](#)
-traditional (traditional C compilation) compiler switch [1-21](#)
transformations [3-6](#)
-trdforinit (traditional initialization) C++ mode compiler switch [1-51](#)
true (*see* Boolean type support keywords (bool, true, false))
twiddle factors for radix-2 FFT [3-94](#)
twiddle factors for radix-4 FFT [3-96](#)
twidfft2d function [3-98](#)
twidfftrad2 function [3-94](#)
twidfftrad4 function [3-96](#)
type conversions [1-71](#)
-typename (support typename) C++ mode compiler switch [1-51](#)

U

-U (undefine macro) compiler switch [1-25](#), [1-42](#)
-unsigned-char (make char unsigned) compiler switch [1-42](#)
upper case (*see* isupper, toupper functions)
user header files [1-105](#)

V

-v (version and verbose) compiler switch [1-42](#)
va_arg (get next argument in variable list) function [2-147](#)
va_end (reset variable list pointer) function [2-149](#)
va_start (set variable list pointer) function [2-150](#)
-val-global (add global names) compiler switch [1-43](#)
var (variance) function [3-100](#)
variable length array support [1-76](#)
VCSE components [1-43](#)
-verbose (display command line) compiler switch [1-43](#)
-version (display version) switch [1-43](#)
VIDL source text [1-107](#)
VisualDSP++ compiler C/C++ language extensions [1-54](#), [1-55](#)
VisualDSP++ Kernel (VDK). [1-42](#)
volatile and asm() C program constructs [1-66](#)
volatile keyword [1-21](#)

W

-w (disable all warnings) switch [1-43](#)
-warn-protos (prototypes warning) compiler switch [1-45](#)
-wchar (support wide char type) C++ mode compiler switch [1-51](#)

INDEX

- Wdriver-limit (maximum process errors) compiler switch [1-44](#)
- weak_entry pragma [1-101](#), [1-102](#)
- Werror-limit (maximum compiler errors) compiler switch [1-44](#)
- white space character test (*see* isspace function)
- workaround compiler switch [1-45](#)
- Wremarks (enable diagnostic warnings) compiler switch [1-44](#)
- write-files (enable driver I/O redirection) compiler switch [1-45](#)
- write-opts compiler switch [1-45](#)

- writing preprocessor macros
 - macros
 - writing preprocessor [1-105](#)
- Wterse (enable terse warnings) compiler switch [1-44](#)

X

- xref (cross-reference list) compiler switch [1-45](#)

Z

- zero_cross (count zero crossing) function [3-101](#)