

VISUAL***DSP++***[®] **3.5** **C/C++ Compiler and Library Manual** **for TigerSHARC[®] Processors**

Revision 1.1, March 2004

Part Number
82-000336-03

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



Copyright Information

©2004 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, the CROSSCORE logo, VisualDSP++, SHARC, TigerSHARC, and EZ-KIT Lite are registered trademarks of Analog Devices, Inc.

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

CONTENTS

PREFACE

Purpose	xxix
Intended Audience	xxix
Manual Contents Description	xxx
What's New in this Manual	xxx
Technical or Customer Support	xxx
Supported Processors	xxxii
Product Information	xxxii
MyAnalog.com	xxxii
Processor Product Information	xxxiii
Related Documents	xxxiv
Online Technical Documentation	xxxiv
From VisualDSP++	xxxv
From Windows	xxxv
From the Web	xxxvi
Printed Manuals	xxxvi
VisualDSP++ Documentation Set	xxxvi

CONTENTS

Hardware Tools Manuals	xxxvii
Processor Manuals	xxxvii
Data Sheets	xxxvii
Notation Conventions	xxxviii

COMPILER

C/C++ Compiler Overview	1-2
Compiler Command-Line Interface	1-4
Running the Compiler	1-5
Specifying Compiler Options in VisualDSP++	1-9
Compiler Command-Line Switches	1-11
C/C++ Compiler Switch Summaries	1-11
C/C++ Mode Selection Switch Descriptions	1-21
-c89	1-21
-c++	1-21
C/C++ Compiler Common Switch Descriptions	1-22
sourcefile	1-22
-@ filename	1-22
-A name [(<tokens>)]<="" td=""><td>1-22</td></tokens>)]>	1-22
-align-branch-lines	1-23
-alttok	1-23
-bss	1-24
-build-lib	1-24
-C	1-24
-c	1-24

-char-size-any	1-24
-char-size-{8 32}	1-25
-const-read-write	1-25
-Dmacro[=definition]	1-25
-debug-types	1-26
-default-branch-{np p}	1-26
-default-linkage-{asm C C++}	1-26
-double-size-any	1-27
-double-size-{32 64}	1-27
-dry	1-28
-dryrun	1-28
-E	1-28
-ED	1-29
-EE	1-29
-extra-keywords	1-29
-flags-{asm compiler lib link mem} switch [,switch2 [,...]]	1-29
-force-circbuf	1-30
-fp-associative	1-30
-full-version	1-30
-g	1-30
-H	1-31
-HH	1-31
-h[elp]	1-31
-I-	1-31

CONTENTS

-I directory [{, ;} directory...]	1-32
-include filename	1-32
-ipa	1-33
-L directory [{, ;} directory...]	1-33
-l library	1-33
-M	1-33
-MD	1-34
-MM	1-34
-Mo filename	1-34
-Mt filename	1-34
-MQ	1-34
-map filename	1-35
-mem	1-35
-no-align-branch-lines	1-35
-no-alttok	1-35
-no-annotate	1-35
-no-bss	1-36
-no-builtin	1-36
-no-circbuf	1-36
-no-const-strings	1-36
-no-defs	1-37
-no-extra-keywords	1-37
-no-fp-associative	1-37
-no-mem	1-37

-no-saturation	1-37
-no-std-ass	1-38
-no-std-def	1-38
-no-std-inc	1-38
-no-std-lib	1-38
-no-threads	1-38
-O	1-39
-O[0 1]	1-39
-Oa	1-39
-Os	1-39
-Ov num	1-40
-o filename	1-40
-P	1-40
-PP	1-40
-path-{ asm compiler def lib link mem } directory	1-40
-path-install directory	1-41
-path-output directory	1-41
-path-temp directory	1-41
-pch	1-41
-pchdir directory	1-42
-pedantic	1-42
-pedantic-errors	1-42
-pguide	1-42
-pplist filename	1-42

CONTENTS

-proc processor	1-43
-R directory [{: ,}directory ...]	1-44
-R-	1-44
-S	1-44
-s	1-45
-save-temps	1-45
-show	1-45
-si-revision version	1-45
-signed-bitfield	1-46
-signed-char	1-46
-syntax-only	1-46
-sysdefs	1-47
-T filename	1-47
-threads	1-48
-time	1-48
-Umacro	1-48
-unsigned-bitfield	1-48
-unsigned-char	1-49
-v	1-49
-val-global <name-list>	1-49
-vcse-add <filename>	1-50
-vcse-cname <component name>	1-50
-vcse-enforce	1-50
-vcse-names <filename>	1-50

-verbose	1-50
-version	1-51
-W {error remark suppress warn} number	1-51
-Wdriver-limit number	1-51
-Werror-limit number	1-51
-Wremarks	1-51
-Wterse	1-52
-w	1-52
-warn-protos	1-52
-workaround <workaround>[,<workaround>]*	1-52
-write-files	1-52
-write-opts	1-53
-xref <filename>	1-53
C++ Mode Compiler Switch Descriptions	1-54
-anach	1-54
-eh	1-55
-no-anach	1-56
-no-demangle	1-56
-no-eh	1-56
-no-rtti	1-56
-rtti	1-56
Data Types and Data Type Sizes	1-57
Integer Data Types	1-58
Floating-Point Data Types	1-58

CONTENTS

Data Type Alignment	1-59
Optimization Control	1-60
Interprocedural Analysis	1-62
Interaction with Libraries	1-63
C/C++ Compiler Language Extensions	1-66
Byte Addressing Mode	1-70
sizeof Operator Types and Sizes	1-70
Pointers	1-71
Alignment of Objects	1-71
Initializations	1-72
Pragmas Used in Byte Addressing Mode	1-72
Performance Issues	1-72
Libraries Used in Byte Addressing Mode	1-73
Include Files	1-73
Inline Function Support Keyword (inline)	1-74
Inline Assembly Language Support Keyword (asm)	1-75
Assembly Construct Template	1-76
ASM() Construct Syntax:	1-76
ASM() Construct Syntax Rules	1-77
ASM() Construct Template Example	1-78
Assembly Construct Operand Description	1-79
Assembly Constructs with Multiple Instructions	1-85
Assembly Construct Reordering and Optimization	1-85
Assembly Constructs with Input and Output Operands	1-86

Assembly Constructs and Flow Control	1-87
64-Bit Integer Support (long long)	1-87
Quad Word Support	1-88
Memory Support Keywords (pm dm)	1-88
Memory Keyword Rules	1-88
__regclass Construct	1-90
Bank Type Qualifier	1-91
Placement Support Keyword (section)	1-92
Boolean Type Support Keywords	1-93
Pointer Class Support Keyword (restrict)	1-93
Variable Length Array Support	1-94
Non-Constant Aggregate Initializer Support	1-95
Indexed Initializer Support	1-96
Compiler Built-In Functions	1-98
Using the builtins.h Header File	1-99
Optimization Guidance Built-in Functions	1-101
16-Bit Data Types	1-101
Packed 16-bit Integer Support Using C	1-103
Constructors (int2x16 values)	1-103
Extractors and Expanders (int2x16 values)	1-103
Arithmetic Operators	1-104
Bitwise Operators (int2x16 values)	1-105
Comparison Operators (int2x16 values)	1-105
Sideways Sum (int2x16 values)	1-106

CONTENTS

Constructors (int4x16 values)	1-107
Extractors (2x16 from a 4x16 value)	1-108
Arithmetic Operators (int4x16 values)	1-109
Sideways Sum (int4x16 values)	1-110
32-Bit Data Types	1-111
Constructors (int2x32 values)	1-111
Extractors (int2x32 values)	1-112
Circular Buffer Built-In Functions	1-112
Circular Buffer Increment of an Index	1-112
Circular Buffer Increment of a Pointer	1-113
Math Intrinsics	1-114
Instructions Generated by Built-in Functions	1-115
Data Alignment Buffer (DAB) Built-in Functions	1-129
Circular Buffer Data Alignment Buffer (DAB) Built-in Functions 1-131	
Communications Logic Unit Operations	1-132
TMAX, TMAX_ADD, TMAX_SUB, MAX_ADD, MAX_SUB 1-133	
PERMUTE	1-138
ACS	1-139
DESPREAD	1-143
XCORRS	1-145
Pragmas	1-148
Data Alignment Pragmas	1-150
#pragma align num	1-150

Interrupt Handler Pragas	1-151
Loop Optimization Pragas	1-154
#pragma all_aligned	1-154
#pragma no_vectorization	1-154
#pragma loop_count(min, max, modulo)	1-154
#pragma different_banks	1-155
#pragma vector_for	1-155
#pragma no_alias	1-156
Function Side-Effect Pragas	1-157
#pragma pure	1-157
#pragma const	1-158
#pragma regs_clobbered string	1-158
General Optimization Pragas	1-162
Linking Control Pragas	1-163
#pragma linkage_name identifier	1-163
#pragma retain_name	1-164
#pragma weak_entry	1-164
Template Instantiation Pragas	1-165
#pragma instantiate instance	1-166
#pragma do_not_instantiate instance	1-166
#pragma can_instantiate instance	1-166
Preprocessor Generated Warnings	1-167
Increments and Decrements	1-167
C++ Style Comments	1-167

CONTENTS

C++ Fractional Type Support	1-168
Format of Fractional Literals	1-169
Conversions Involving Fractional Values	1-169
Fractional Arithmetic Operations	1-169
Mixed Mode Operations	1-170
GCC Compatibility Extensions	1-171
Statement Expressions	1-171
Type Reference Support Keyword (typeof)	1-172
GCC Generalized Lvalues	1-173
Conditional Expressions with Missing Operands	1-174
Hexadecimal Floating-Point Numbers	1-174
Zero Length Arrays	1-175
Variable Argument Macros	1-175
Line Breaks in String Literals	1-176
Arithmetic on Pointers to Void and Pointers to Functions	1-176
Cast to Union	1-176
Ranges in Case Labels	1-176
Declarations Mixed with Code	1-177
Escape Character Constant	1-177
Alignment Inquiry Keyword (__alignof__)	1-177
Keyword for Specifying Names in Generated Assembler (asm)	1-178
Function, Variable and Type Attribute Keyword (__attribute__)	1-178
Preprocessor Features	1-179
Predefined Preprocessor Macros	1-179

Header Files	1-181
Writing Macros	1-182
Preprocessing of .IDL Files	1-184
C/C++ Run-Time Model and Environment	1-185
Stack Frame Overview	1-185
Stack Frame Description	1-188
General System-Wide Specifications	1-189
At a procedure call, the following must be true:	1-190
Argument Passage	1-191
Return Values	1-193
Procedure Call and Return	1-193
To Call a Procedure:	1-193
On Entry:	1-194
To Return from a Procedure:	1-195
Code Sequences	1-195
Support for argv/argc	1-197
File I/O Support	1-197
Extending I/O Support To New Devices	1-198
Using Multiple Heaps	1-200
Heap Identifiers	1-201
Initializing the Heap	1-202
Using Alternate Heaps with the Standard Interface	1-202
Using the Alternate Heap Interface	1-202
Miscellaneous Information	1-204

CONTENTS

Register Classification	1-204
Callee Preserved Registers (“Preserved”)	1-204
Caller Save Registers (“Scratch”)	1-204
ADSP-TS101 and ADSP-TS20x Processor Registers	1-205
C/C++ and Assembly Language Interface	1-213
Calling Assembly Subroutines from C/C++ Programs	1-213
Calling C/C++ Functions from Assembly Programs	1-216
Using Mixed C/C++ and Assembly Naming Conventions	1-217
C++ Programming Examples	1-219
Using Fract Type Support	1-219
Using Complex Number Support	1-220
ACHIEVING OPTIMAL PERFORMANCE FROM C/C++ SOURCE CODE	
General Guidelines	2-3
How the Compiler Can Help	2-4
Using the Compiler Optimizer	2-4
Using Profile-Guided Optimization	2-5
Using Interprocedural Optimization	2-6
Data Types	2-7
Avoiding Emulated Arithmetic	2-8
Using Sub-Word Types with Caution	2-9
Getting the Most from IPA	2-10
Initialize Constants Staticly	2-10
Quad-Word-Aligning Your Data	2-11

Using <code>__builtin_aligned</code>	2-12
Avoiding Aliases	2-14
Indexed Arrays vs. Pointers	2-15
Trying Pointer and Indexed Styles	2-16
Function Inlining	2-17
Using Inline asm Statements	2-18
Memory Usage	2-19
Putting Arrays into Different Memory Sections	2-19
Using the Bank Qualifier	2-21
Loop Guidelines	2-23
Keeping Loops Short	2-23
Avoiding Unrolling Loops	2-23
Avoiding Loop-Carried Dependencies	2-24
Avoiding Loop Rotation by Hand	2-25
Avoiding Array Writes in Loops	2-26
Inner Loops vs. Outer Loops	2-27
Avoiding Conditional Code in Loops	2-27
Avoiding Placing Function Calls in Loops	2-28
Avoiding Non-Unit Strides	2-29
Loop Control	2-30
Using the Restrict Qualifier	2-31
Using the Const Qualifier	2-31
Avoiding Long Latencies	2-32
Using Built-In Functions in Code Optimization	2-33

CONTENTS

Using Fractional Data	2-33
System Support Built-in Functions	2-34
Using Circular Buffers	2-35
Smaller Applications: Optimizing for Code Size	2-37
Pragmas	2-39
Function Pragmas	2-39
#pragma const	2-39
#pragma pure	2-40
#pragma regs_clobbered	2-40
#pragma optimize_{off for_speed for_space}	2-42
Loop Optimization Pragmas	2-42
#pragma loop_count	2-42
#pragma no_vectorization	2-43
#pragma vector_for	2-43
#pragma all_aligned	2-44
#pragma different_banks	2-46
#pragma no_alias	2-46
Useful Optimization Switches	2-47
Example: How the Optimizer Works	2-48
Assembly Optimizer Annotations	2-51
Procedure Statistics	2-52
Loop Identification	2-54
Loop Identification Annotations	2-55
File Position	2-58

Vectorization	2-60
Unroll and Jam	2-61
Loop Flattening	2-63
Vectorization Annotations	2-64
Modulo Scheduling	2-66

C/C++ RUN-TIME LIBRARY

C and C++ Run-Time Libraries Guide	3-3
Calling Library Functions	3-3
Using Compiler's Built-In C Library Functions	3-4
Linking Library Functions	3-5
Working With Library Source Code	3-7
Working With Library Header Files	3-8
assert.h	3-10
ctype.h	3-10
errno.h	3-10
float.h	3-10
iso646.h	3-11
limits.h	3-11
locale.h	3-12
math.h	3-12
setjmp.h	3-13
signal.h	3-13
stdarg.h	3-14
stddef.h	3-14

CONTENTS

stdio.h	3-14
Default Device Driver Interface	3-17
Data Packing For Primitive I/O	3-18
Data Structure for Primitive I/O	3-19
stdlib.h	3-21
string.h	3-22
time.h	3-22
DSP Header Files	3-22
complex.h — Basic Complex Arithmetic Functions	3-23
filter.h — DSP Filters and Transformations	3-24
libsim.h — Simulator Services	3-25
matrix.h — Matrix Functions	3-26
stats.h — Statistical Functions	3-27
vector.h — Vector Functions	3-28
window.h — Window Generators	3-29
Abridged C++ Library Support	3-30
Embedded C++ Library Header Files	3-30
complex	3-30
exception	3-31
fract	3-31
fstream	3-31
iomanip	3-31
ios	3-31
iosfwd	3-31

iostream	3-31
istream	3-32
new	3-32
ostream	3-32
sstream	3-32
stdexcept	3-32
streambuf	3-32
string	3-32
strstream	3-33
C++ Header Files for C Library Facilities	3-33
Embedded Standard Template Library Header Files	3-34
algorithm	3-34
deque	3-34
functional	3-34
hash_map	3-34
hash_set	3-35
iterator	3-35
list	3-35
map	3-35
memory	3-35
numeric	3-35
queue	3-35
set	3-35
stack	3-35

CONTENTS

utility	3-36
vector	3-36
fstream.h	3-36
iomanip.h	3-36
iostream.h	3-36
new.h	3-36
Documented Library Functions	3-37
Run-Time Library Reference	3-41
Notation Conventions	3-41
Reference Format	3-41
a_compress	3-42
a_expand	3-43
abs	3-44
acos	3-45
addbitrev	3-46
alog	3-47
alog10	3-49
arg	3-51
asin	3-52
atan	3-53
atan2	3-54
atof	3-55
atoi	3-57
atol	3-58

atold	3-59
atoll	3-61
autocoh	3-62
autocorr	3-63
avg	3-64
bsearch	3-65
cabs	3-67
cadd	3-68
cartesian	3-69
cdiv	3-71
ceil	3-72
cexp	3-73
cfft	3-74
cfft_mag	3-77
cfft2d	3-79
cfft_f	3-81
clip	3-83
cmatmadd	3-84
cmatmmlt	3-85
cmatmsub	3-86
cmatsadd	3-87
cmatsmlt	3-88
cmatssub	3-89
cmlt	3-90

conj	3-91
convolve	3-92
conv2d	3-93
copysign	3-94
cos	3-95
cosh	3-96
cot	3-97
count_ones	3-98
crosscoh	3-99
crosscorr	3-100
csub	3-101
cvecdot	3-102
cvecsadd	3-103
cvecsmult	3-104
cvecssub	3-105
cvecvadd	3-106
cvecvmult	3-107
cvecvsub	3-108
div	3-109
exp	3-110
__emuclk	3-111
fabs	3-112
favg	3-113
fclip	3-114

<code>fir</code>	3-115
<code>fir_decima</code>	3-117
<code>fir_interp</code>	3-119
<code>floor</code>	3-124
<code>fmax</code>	3-125
<code>fmin</code>	3-126
<code>fmod</code>	3-127
<code>frexp</code>	3-128
<code>gen_bartlett</code>	3-129
<code>gen_blackman</code>	3-131
<code>gen_gaussian</code>	3-132
<code>gen_hamming</code>	3-133
<code>gen_hanning</code>	3-134
<code>gen_harris</code>	3-135
<code>gen_kaiser</code>	3-136
<code>gen_rectangular</code>	3-137
<code>gen_triangle</code>	3-138
<code>gen_vonhann</code>	3-140
<code>histogram</code>	3-141
<code>ifft</code>	3-142
<code>ifft2d</code>	3-144
<code>iir</code>	3-146
<code>interrupt, interruptf, interrupts</code>	3-148
<code>log</code>	3-152

log10	3-153
matinv	3-154
matmadd	3-155
matmmlt	3-156
matmsub	3-157
matsadd	3-158
matsmult	3-159
matssub	3-160
max	3-161
mean	3-162
min	3-163
modf	3-164
mu_compress	3-165
mu_expand	3-166
norm	3-167
polar	3-168
pow	3-169
qsort	3-170
raise	3-172
rand	3-175
rfft	3-176
rfft_mag	3-179
rfft2d	3-181
rfftf	3-183

rfftf_mag	3-185
rms	3-187
rsqrt	3-188
sign	3-189
signal, signalf, signals	3-190
sin	3-193
sinh	3-194
sqrt	3-195
srand	3-196
strtod	3-197
strtof	3-199
strtoi	3-201
strtol	3-202
strtold	3-203
strtoll	3-205
strtoul	3-206
strtoull	3-207
tan	3-209
tanh	3-210
transpm	3-211
twidfft	3-212
twidfft_f	3-214
var	3-216
vecdot	3-217

vecsadd	3-218
vecsmult	3-219
vecssub	3-220
vecvadd	3-221
vecvmult	3-222
vecvsub	3-223
zero_cross	3-224

INDEX

PREFACE

Thank you for purchasing Analog Devices, Inc. development software for digital signal processor (DSP) applications.

Purpose

The *VisualDSP++ 3.5 C/C++ Compiler and Library Manual for TigerSHARC Processors* contains information about the C/C++ compiler and run-time library for TigerSHARC® (ADSP-TSxxx) processors. 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 a programmer who is familiar with Analog Devices processors. This manual assumes that the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts (such as the appropriate hardware reference and programming reference manuals) that describe your target architecture.

Manual Contents Description

This manual contains:

- Chapter 1, “[Compiler](#)”
Provides information on compiler options, language extensions and C/C++/assembly interfacing
- Chapter 2, “[Achieving Optimal Performance from C/C++ Source Code](#)”
Provides information on compiler (and assembly) code optimization (techniques and options).
- Chapter 3, “[C/C++ Run-Time Library](#)”
Shows how to use library functions and provides a complete C/C++ library function reference (for functions covered in the current compiler release)

What's New in this Manual

This edition of the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual for TigerSHARC Processors* documents support for all currently available TigerSHARC processors listed in “[Supported Processors](#)”.

Refer to the *VisualDSP++ 3.5 Product Release Bulletin for 32-Bit Processors* for a complete list of new compiler features and enhancements.

Technical or Customer Support

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

- Visit the Embedded Processing and DSP products Web site at <http://www.analog.com/processors/technicalSupport>
- E-mail tools questions to dsptools.support@analog.com
- E-mail processor questions to dsp.support@analog.com
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:

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

Supported Processors

The name “TigerSHARC” refers to a family of Analog Devices, Inc. floating-point and [8-bit, 16-bit, and 32-bit] fixed-point processors. VisualDSP++ currently supports the following TigerSHARC processors:

- ADSP-TS101 DSP
- ADSP-TS201 DSP
- ADSP-TS202 DSP
- ADSP-TS203 DSP

Product Information

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

Analog Devices is online at www.analog.com. Our Web site 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 Web site that allows customization of a Web page to display only the latest information on products you are interested in. You can also choose to receive weekly E-mail 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 E-mail address.

Processor Product Information

For information on embedded processors and DSPs, visit our Web site at www.analog.com/processors, which provides access to technical publications, data sheets, 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.

- E-mail 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 FTP Web site at
[ftp ftp.analog.com](ftp://ftp.analog.com) or [ftp 137.71.23.21](ftp://137.71.23.21)
<ftp://ftp.analog.com>

Related Documents

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

VisualDSP++ 3.5 Getting Started Guide for 32-Bit Processors

VisualDSP++ 3.5 User's Guide for 32-Bit Processors

VisualDSP++ 3.5 C/C++ Assembler and Preprocessor Manual for TigerSHARC Processors

VisualDSP++ 3.5 Linker and Utilities Manual for 32-Bit Processors

VisualDSP++ 3.5 Product Release Bulletin for 32-Bit Processors

VisualDSP++ Kernel (VDK) User's Guide

VisualDSP++ Component Software Engineering User's Guide

Quick Installation Reference Card

For hardware information, refer to your processors's hardware reference, programming reference, and data sheet. All documentation is available online. Most documentation is available in printed form.

Visit the Technical Library Web site at

<http://www.analog.com/processors/resources/technicalLibrary>
to view all processor and tools manuals and data sheets.

Online Technical Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library and Flexible License Manager (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 of most manuals are also provided.

Each documentation file type is described in the following table.

File	Description
.CHM	Help system files and manuals in Help format
.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 by running the Tools installation. Access the online documentation from the VisualDSP++ environment, Windows[®] Explorer, or the Analog Devices Web site.

From VisualDSP++

From VisualDSP++ environment:

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

Product Information

Using Windows Explorer

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

Using the Windows Start Button

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

From the Web

Download manuals at the following Web site:

<http://www.analog.com/processors/resources/technicalLibrary/manuals>

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

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

To purchase VisualDSP++ manuals, call 1-603-883-2430. The manuals may be purchased only as a kit.

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

Hardware Tools Manuals

To purchase EZ-KIT Lite™ and In-Circuit Emulator (ICE) manuals, call **1-603-883-2430**. The manuals may be ordered by title or by product number located on the back cover of each manual.

Processor Manuals

Hardware reference and instruction set reference manuals may be ordered through the Literature Center at **1-800-ANALOGD (1-800-262-5643)**, or downloaded from the Analog Devices Web site. Manuals may be ordered by title or by product number located on the back cover of each manual.

Data Sheets

All data sheets (preliminary and production) may be downloaded from the Analog Devices Web site. Final data sheets (Rev. 0, A, B, C and so on) can be obtained from the Literature Center at **1-800-ANALOGD (1-800-262-5643)** or downloaded from the Web site.

To have a data sheet faxed to you, call **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. Call the Literature Center first to find out if requested data sheets are available.

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. Code has been formatted to fit this manual's page width.

Example	Description
Close command (File menu)	Titles in reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the Close command appears on the File menu).
{this that}	Alternative items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as <i>this</i> or <i>that</i> . One or the other is required.
[this that]	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional <i>this</i> or <i>that</i> .
[this,...]	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of <i>this</i> .
.SECTION	Commands, directives, keywords, and feature names are in text with letter gothic font.
<i>filename</i>	Non-keyword placeholders appear in text with italic style format.
	This symbol indicates a note that provides supplementary information on a related topic. In the online version of this book, the word Note appears instead of this symbol.
	This symbol indicates a warning that advises on an inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word Warning appears instead of this symbol.

1 COMPILER

The C/C++ compiler (`ccts`) is part of Analog Devices development software for TigerSHARC (ADSP-TSxxx) processors.

This chapter contains:

- [“C/C++ Compiler Overview” on page 1-2](#)
Provides an overview of C/C++ compiler for TigerSHARC processors.
- [“Compiler Command-Line Interface” on page 1-4](#)
Describes the operation of the compiler as it processes programs, including input and output files, and command-line switches.
- [“C/C++ Compiler Language Extensions” on page 1-66](#)
Describes the `ccts` compiler’s extensions to the ISO/ANSI standard for the C and C++ languages.
- [“Preprocessor Features” on page 1-179](#)
Contains information on the preprocessor and ways to modify source compilation.
- [“C/C++ Run-Time Model and Environment” on page 1-185](#)
Contains reference information about implementation of C/C++ programs, data, and function calls in TigerSHARC processors
- [“C/C++ and Assembly Language Interface” on page 1-213](#)
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 (`ccts`) is designed to aid your DSP project development efforts by:

- Processing C and C++ source files, producing machine level versions of the source code and object files
- Providing relocatable code and debugging information within the object files
- Providing relocatable data and program memory segments for placement by the linker in the processors' memory

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

The C/C++ compiler (`ccts`) compiles ISO/ANSI standard C and C++ code for the TigerSHARC processors. Additionally, Analog Devices includes within the compiler a number of C language extensions designed to assist in DSP development. The `ccts` compiler runs from the VisualDSP++ environment or from an operating system command line.

The C/C++ compiler processes your C and C++ language source files and produces TigerSHARC assembler source files. The assembler source files are assembled by the TigerSHARC assembler (`easmts`). The assembler creates Executable and Linkable Format (ELF) object files that can either be linked (using the linker) to create an 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 `ccts` 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 `ccts` compiler supports a set of C/C++ language extensions. These extensions support hardware features of the TigerSHARC processors. For more information, see “[C/C++ Compiler Language Extensions](#)” on [page 1-66](#).

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 1-9](#)). The selections control how the compiler processes your source files, letting you select features that include the language dialect, error reporting, and debugger output, etc.

By default, the `ccts` compiler operates in the 32-bit word addressing mode. The `ccts` compiler can also be set for the 8-bit byte addressing mode. For more information, refer to “[-char-size-any](#)” switch (on [page 1-24](#)), “[-char-size-{8|32}](#)” switch (on [page 1-25](#)), “[Data Types and Data Type Sizes](#)” on [page 1-57](#), and “[Byte Addressing Mode](#)” on [page 1-70](#).

For more information on the VisualDSP++ environment, see the *VisualDSP++ 3.5 User’s Guide for TigerSHARC Processors* and online Help.

Compiler Command-Line Interface

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

This section contains:

- [“Running the Compiler” on page 1-5](#)
- [“Specifying Compiler Options in VisualDSP++” on page 1-9](#)
- [“Compiler Command-Line Switches” on page 1-11](#)
- [“Data Types and Data Type Sizes” on page 1-57](#)
- [“Data Type Alignment” on page 1-59](#)
- [“Optimization Control” on page 1-60](#)

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 1-6](#) 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 1-8](#)). [Table 1-3 on page 1-11](#) 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 1-12](#). A summary of the C++ specific compiler switches appears in [Table 1-5 on page 1-20](#). The summaries are followed by descriptions of each switch.



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

Running the Compiler

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

```
ccts [-switch [-switch ...]] sourcefile [sourcefile ...]
```

runs the assembler with

<code>ccts</code>	Name of the compiler program for TigerSHARC processors.
<code>-switch</code>	Switch (or switches) to process. The compiler has many switches. These switches select the operations and modes for the compiler and other tools. Command-line switches are case sensitive. For example, <code>-O</code> is not the same as <code>-o</code> .
<code>sourceFile</code>	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 `ccts` compiler uses the file extension to determine what the file contains and what operations to perform upon it. [Table 1-2 on page 1-8](#) lists the allowed extensions.

For example, the following command line

```
ccts -proc ADSP-TS101 -O -Wremarks -o program.dxe source.c
```

runs `ccts` with the following switches:

<code>-proc ADSP-TS101</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

Compiler Command-Line Interface

`-o program.dxe` Selects a name for the compiled, linked output
`source.c` Specifies the C language source file to be
 compiled

The following example command line

```
ccts -proc ADSP-TS101 -c++ source.cpp
```

runs `ccts` 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 `ccts` 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 .cc .cpp .cxx</code>	Source file is compiled, assembled, and linked
<code>.asm, .dsp, or .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, `ccts` names the output for you. [Table 1-2 on page 1-8](#) lists the type of files, names, and extensions `ccts` 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 1-32](#)) and `-L` directory switch ([on page 1-33](#)).

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 `ccts` 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, `ccts` 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.
.cc .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.
.ipa, .opa	Interprocedural analysis files — used internally by the compiler when performing interprocedural analysis.
.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.
.xml	Processor system memory map file output.
.sym	Processor system symbol map file output.

Specifying Compiler Options in VisualDSP++

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

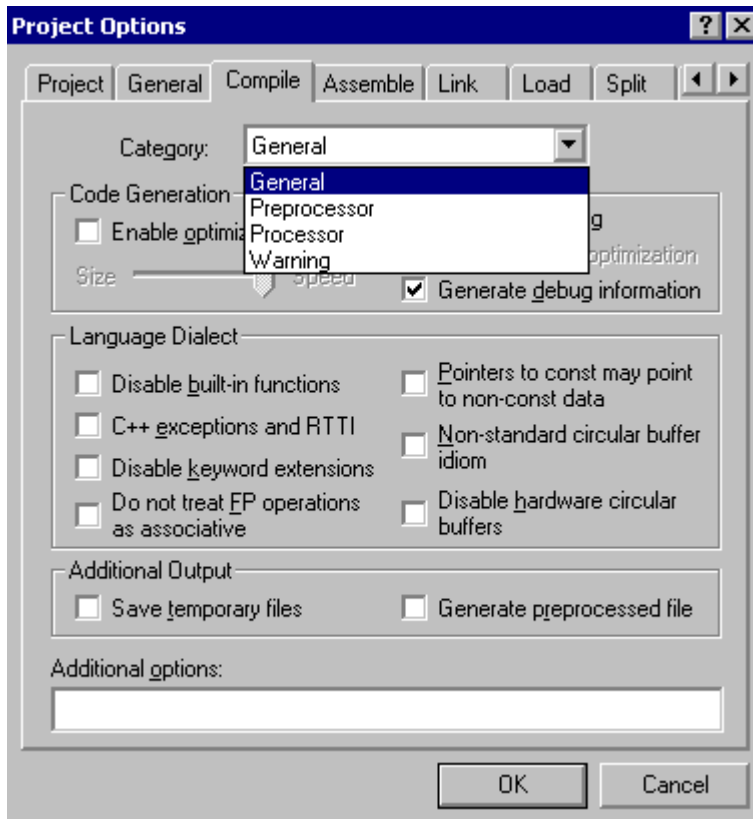


Figure 1-1. Project Options – Compile General Property Page

Compiler Command-Line Interface

There are four sub-pages you can access—**General**, **Preprocessor**, **Processor**, and **Warning**. Most project options have a corresponding compiler command-line switch described in [“Compiler Command-Line Switches” on page 1-11](#). The **Additional options** field in each sub-page is used to enter the appropriate file names and options that do not have corresponding controls on the **Compile** tab but are available as compiler switches.

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:

- [Table 1-3 “C or C++ Mode Selection Switches” on page 1-11](#)
- [Table 1-4 “C/C++ Compiler Common Switches” on page 1-12](#)
- [Table 1-5 “C++ Mode Compiler Switches” on page 1-20](#)

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
-c89 (on page 1-21)	Supports programs that conform to the ISO/IEC 9899:1990 standard
-c++ (on page 1-21)	Supports ANSI/ISO standard C++ with Analog Devices extensions.

Compiler Command-Line Interface

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>-A name[tokens]</code> (on page 1-22)	Asserts the specified name as a predicate.
<code>-align-branch-lines</code> (on page 1-23)	Quad align predicted branches.
<code>-alttok</code> (on page 1-23)	Allows alternative keywords and sequences in sources.
<code>-bss</code> (on page 1-24)	Causes the compiler to place global zero-initialized data into a separate BSS-style section.
<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>-char-size-any</code> (on page 1-24)	Indicate that the resulting object can link against any <code>char</code> size object.
<code>-char-size-{8 32}</code> (on page 1-25)	Selects byte (8-bit) or word (32-bit) addressing mode.
<code>-const-read-write</code> (on page 1-25)	Constant pointers may access modifiable memory.
<code>-Dmacro[=definition]</code> (on page 1-25)	Defines a macro.
<code>-debug-types</code> (on page 1-26)	Supports building a <code>*.h</code> file directly and writing a complete set of debugging information for the header file.
<code>-default-branch-{np p}</code> (on page 1-26)	Sets default branches to be predict or non-predict.

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

Switch Name	Description
<code>-default-linkage-{asm C C++}</code> (on page 1-26)	Sets the default linkage type (C, C++, asm).
<code>-double-size-any</code> (on page 1-27)	Indicate that the resulting object can link against any double size object.
<code>-double-size-{32 64}</code> (on page 1-27)	Selects 32- or 64-bit IEEE format for double. The <code>-double-size-32</code> is the default mode.
<code>-dry</code> (on page 1-28)	Displays, but does not perform, main driver actions (verbose dry-run).
<code>-dryrun</code> (on page 1-28)	Displays, but does not perform, top-level driver actions (terse dry-run).
<code>-E</code> (on page 1-28)	Preprocesses, but does not compile, the source file.
<code>-ED</code> (on page 1-29)	Preprocesses and sends all output to a file
<code>-EE</code> (on page 1-29)	Preprocesses and compiles the source file.
<code>-extra-keywords</code> (on page 1-29)	Recognizes Analog Devices extensions to ISO/ANSI standards for C and C++. Default mode.
<code>-flags-{tools} <arg1> [,arg2...]</code> (on page 1-29)	Passes command-line switches through the compiler to other build tools.
<code>-force-circbuf</code> (on page 1-30)	Treats array references of the form <code>array[i%n]</code> as circular buffer operations
<code>-force-link</code> (on page 1-30)	Forces stack frame creation for leaf functions. Always creates a new stack frame for leaf functions (defaults to ON with <code>-g</code> option set, enforced for the <code>-p</code> option)
<code>-fp-associative</code> (on page 1-30)	Treats floating point multiply and addition as an associative.
<code>-full-version</code> (on page 1-30)	Displays version information for build tools.

Compiler Command-Line Interface

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

Switch Name	Description
<code>-g</code> (on page 1-30)	Generates DWARF-2 debug information.
<code>-H</code> (on page 1-31)	Outputs a list of header files, but does not compile the source file.
<code>-HH</code> (on page 1-31)	Outputs a list of included header files and compiles.
<code>-h[elp]</code> (on page 1-31)	Outputs a list of command-line switches with brief syntax descriptions.
<code>-I-</code> (on page 1-31)	Establishes the point in the <code>include</code> directory list at which the search for header files enclosed in angle brackets should begin.
<code>-I directory</code> (on page 1-31)	Appends <i>directory</i> to the standard search path.
<code>-include filename</code> (on page 1-32)	Includes named file prior to preprocessing each source file.
<code>-ipa</code> (on page 1-33)	Enables interprocedural analysis.
<code>-L directory</code> (on page 1-33)	Appends the specified directory to the standard library search path when linking.
<code>-l library</code> (on page 1-33)	Searches the specified library for functions when linking.
<code>-M</code> (on page 1-33)	Generates make rules only; does not compile.
<code>-MD</code> (on page 1-34)	Generates make rule, compiles, and prints to a file.
<code>-MM</code> (on page 1-34)	Generates make rules and compiles.
<code>-Mo filename</code> (on page 1-34)	Writes dependency information to <i>filename</i> . This switch is used in conjunction with the <code>-ED</code> or <code>-MD</code> options.

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

Switch Name	Description
-Mt <i>filename</i> (on page 1-34)	Makes dependencies for the specified source file.
-MQ (on page 1-34)	Generates make rules only; does not compile. No notification when input files are missing.
-map <i>filename</i> (on page 1-35)	Directs the linker to generate a memory map of all symbols.
-mem (on page 1-35)	Enables memory initialization.
-no-align-branch-lines (on page 1-35)	Do not align predicted branches to a quad word boundary.
-no-alttok (on page 1-35)	Does not allow alternative keywords and sequences in sources.
-no-annotate (on page 1-35)	Disables the annotation of assembly files.
-no-bss (on page 1-36)	Causes the compiler to group global zero-initialized data into the same section as global data with non-zero initializers. Set by default.
-no-builtin (on page 1-36)	For certain language extensions, uses generic implementations in preference to intrinsic functions. See “ Math Intrinsics ” on page 1-114.
-no-circbuf (on page 1-36)	Disables the automatic generation of circular buffer code by the compiler.
-no-const-strings (on page 1-36)	Indicates that string literals should not be qualified as <code>const</code> .
-no-defs (on page 1-37)	Does not define any default preprocessor macros, include directories, library directories, libraries, run-time headers, or keyword extensions.
-no-extra-keywords (on page 1-37)	Does not define language extension keywords that could be valid C or C++ identifiers.
-no-fp-associative (on page 1-37)	Does not treat floating point multiply and addition as an associative.

Compiler Command-Line Interface

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

Switch Name	Description
<code>-no-mem</code> (on page 1-37)	Disables memory initialization.
<code>-no-saturation</code> (on page 1-37)	Causes the compiler not to introduce saturation semantics when optimizing expressions.
<code>-no-std-ass</code> (on page 1-38)	Disables any predefined assertions and system-specific macro definitions.
<code>-no-std-def</code> (on page 1-38)	Disables normal macro definitions; also disables Analog Devices keyword extensions that do not have leading underscores (<code>__</code>).
<code>-no-std-inc</code> (on page 1-38)	Searches for preprocessor header files only in the current directory and in directories specified with the <code>-I</code> switch
<code>-no-std-lib</code> (on page 1-38)	Searches for only those linker libraries specified with the <code>-l</code> switch when linking.
<code>-no-threads</code> (on page 1-38)	Specifies that compiled code does not need to be thread-safe.
<code>-O [0 1]</code> (on page 1-39)	Enables code optimizations.
<code>-Oa</code> (on page 1-39)	Enables automatic function inlining.
<code>-Os</code> (on page 1-39)	Optimizes for code size.
<code>-Ov num</code> (on page 1-40)	Controls speed vs. size optimizations.
<code>-o filename</code> (on page 1-40)	Specifies the output file name.
<code>-P</code> (on page 1-40)	Preprocesses, but does not compile, the source file. Omits line numbers in the preprocessor output.
<code>-PP</code> (on page 1-40)	Similar to <code>-P</code> , but does not halt compilation after preprocessing.

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

Switch Name	Description
<code>-path-{asm compiler def lib link mem} <i>directory</i></code> (on page 1-40)	Uses the specified directory as the location of the specified compilation tool (assembler, compiler, driver, librarian, linker, or and memory initializer, respectively).
<code>-path-install <i>directory</i></code> (on page 1-41)	Uses the specified directory as the location for all compilation tool.
<code>-path-output <i>directory</i></code> (on page 1-41)	Specifies the location of non-temporary files.
<code>-path-temp <i>directory</i></code> (on page 1-41)	Specifies the location of temporary files.
<code>-pch</code> (on page 1-41)	Generates and uses precompiled header files (*.pch)
<code>-pchdir <i>directory</i></code> (on page 1-41)	Specifies the location of PCHRepository.
<code>-pedantic</code> (on page 1-42)	Issues compiler warnings for any constructs that are not ISO/ANSI standard C/C++-compliant.
<code>-pedantic-errors</code> (on page 1-42)	Issues compiler errors for any constructs that are not ISO/ANSI standard C/C++-compliant.
<code>-pguide</code> (on page 1-42)	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization.
<code>-pplist <i>filename</i></code> (on page 1-42)	Outputs a raw preprocessed listing to the specified file.
<code>-proc <i>processor</i></code> (on page 1-43)	Outputs a raw preprocessed listing to the specified file.
<code>-R <i>directory</i></code> (on page 1-44)	Appends directory to the standard search path for source files.
<code>-R-</code> (on page 1-44)	Removes all directories from the standard search path for source files.
<code>-S</code> (on page 1-44)	Stops compilation before running the assembler.
<code>-s</code> (on page 1-45)	Removes debugging information from the output executable file when linking.

Compiler Command-Line Interface

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

Switch Name	Description
<code>-save-temps</code> (on page 1-45)	Saves intermediate compiler temporary files.
<code>-show</code> (on page 1-45)	Displays the driver command-line information.
<code>-si-revision <i>version</i></code> (on page 1-40)	Specifies a silicon revision of the specified processor. The default setting is the latest silicon revision.
<code>-signed-bitfield</code> (on page 1-46)	Makes the default type for plain <code>int</code> bitfields signed
<code>-signed-char</code> (on page 1-46)	Makes the default type for <code>char</code> signed.
<code>-syntax-only</code> (on page 1-46)	Checks the source code for compiler syntax errors, but does not write any output.
<code>-sysdefs</code> (on page 1-47)	Defines the system definition macros.
<code>-T <i>filename</i></code> (on page 1-47)	Uses the specified the Linker Description File as control input for linking.
<code>-threads</code> (on page 1-48)	Specifies that the build and link should be thread-safe.
<code>-time</code> (on page 1-48)	Displays the elapsed time as part of the output information on each part of the compilation process.
<code>-Umacro</code> (on page 1-48)	Undefines macro (s).
<code>-unsigned-bitfield</code> (on page 1-49)	Makes the default type for plain <code>int</code> bitfields unsigned
<code>-unsigned-char</code> (on page 1-49)	Makes the default type for <code>char</code> unsigned.
<code>-v</code> (on page 1-49)	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 1-49)	Specifies that the names given by the <code>name-list</code> are present in globally defined variables.

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

Switch Name	Description
-vcse-add <filename> (on page 1-50)	If the -vcse-enforce switch has been supplied, <filename> is passed to the vcse_enforce utility as the argument for the -add switch.
-vcse-cname <component name> (on page 1-50)	If the -vcse-enforce switch has been supplied, <component name> is passed to the vcse_enforce utility as the argument for the -cname switch.
-vcse-enforce (on page 1-50)	If the library builder is invoked, ccts invokes the vcse_enforce utility to process the generated library.
-vcse-names <filename> (on page 1-50)	If the -vcse-enforce switch has been supplied, <filename> is passed to the vcse_enforce utility as the argument for the -names switch.
-verbose (on page 1-50)	Displays command-line information for all compilation tools as they process each file.
-version (on page 1-51)	Displays version information for all compilation tools as they process each file.
-W{error remark suppress warn} <i>number</i> (on page 1-51)	Overrides the default severity of the specified diagnostic-messages (errors, remarks, or warnings).
-Wdriver-limit <i>number</i> (on page 1-51)	Aborts the driver after reaching the specified number of errors.
-Werror-limit <i>number</i> (on page 1-51)	Stops compiling after reaching the specified number of errors.
-Wremarks (on page 1-51)	Indicates that the compiler may issue remarks, which are diagnostic messages even milder than warnings.
-Wterse (on page 1-52)	Issues only the briefest form of compiler warnings, errors, and remarks.
-w (on page 1-52)	Does not display compiler warning messages.
-warn-protos (on page 1-52)	Produces a warnings when a function is called without a prototype.

Compiler Command-Line Interface

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

Switch Name	Description
<code>-workaround <workaround></code> (on page 1-52)	Enables code generator workaround for specific hardware defects.
<code>-write-files</code> (on page 1-52)	Enables compiler I/O redirection.
<code>-write-opts</code> (on page 1-53)	Passes the user-specified options (but not the input filenames).
<code>-xref filename</code> (on page 1-53)	Outputs cross-reference information to the specified file.

Table 1-5. C++ Mode Compiler Switches

Switch Name	Description
<code>-anach</code> (on page 1-54)	Supports some language features (anachronisms) that are prohibited by the C++ standard but still in common use
<code>-eh</code> (on page 1-55)	Enables exception handling
<code>-no-anach</code> (on page 1-56)	Disallows the use of anachronisms that are prohibited by the C++ standard
<code>-no-demangle</code> (on page 1-56)	Prevents filtering of any linker errors through the demangler.
<code>-no-eh</code> (on page 1-56)	Disables exception-handling
<code>-no-rtti</code> (on page 1-56)	Disables run-time type information
<code>-rtti</code> (on page 1-56)	Enables run-time type information

C/C++ Mode Selection Switch Descriptions

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

-c89

The `-c89` switch directs the compiler to support programs that conform to the ISO/IEC 9899:1990 standard. For greater conformance to the standard, the following switches should be used: `-alttok`, `-const-read-write`, `no-extra-keywords`, and `-pedantic` (see in [Table 1-4 on page 1-12](#)).

-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. When using this switch, source files with an extension of `.c` will be compiled and linked in C++ mode.

All the standard features of C++ are accepted in the default mode except exception handling and run-time type identification because these impose a run-time overhead that is not desirable for all embedded programs.

Support for these features can be enabled with the `-eh` and `-rtti` switches (see [Table 1-5 on page 1-20](#)).

Exceptions also require modifications to the Linker Description Files; these modifications link against versions of the C++ run-time library that have been built with exceptions support enabled, and link the exception-handling library. There are several new data sections that must be mapped into the DXE as well: `.frt`, `.edt`, `.cht`, and `.gdt`. See the default LDFs included with the release for example modifications.

Compiler Command-Line Interface

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 *ccts* compiler uses the file extension to determine the operations to perform. [Table 1-2 on page 1-8](#) 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.

-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	embedded	
machine	adspts	
cpu	adspts101	for ADSP-TS101 processor
	adspts201	for ADSP-TS201 processor
	adspts202	for ADSP-TS202 processor
	adspts203	for ADSP-TS203 processor
compiler	ccts	

-align-branch-lines

The `-align-branch-lines` switch instructs the assembler to align all instruction lines containing a predicted branch to quad-word boundaries. This is the default.

-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 1-35](#)) 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>
<code>bitor</code>	<code> </code>
<code>compl</code>	<code>~</code>
<code>not</code>	<code>!</code>
<code>not_eq</code>	<code>!=</code>
<code>or</code>	<code> </code>
<code>or_eq</code>	<code> =</code>
<code>xor</code>	<code>^</code>
<code>xor_eq</code>	<code>^=</code>



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

Compiler Command-Line Interface

-bss

The `-bss` switch directs the compiler to place global zero-initialized data into a BSS-style section (called “bsz”), rather than into normal global data section. See also “[-no-bss](#)” on page 1-36.

-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.

-char-size-any

The `-char-size-any` switch indicates that the resulting object file should be marked in such a way that it can link against objects marked with any `char` size (8 bit or 32 bit). When linking a project with modules compiled with 8-bit and 32-bit compilers, use the `-char-size-any` switch to avoid any linking error messages.

-char-size-{8|32}

The `-char-size-{8|32}` switch specifies that `chars` are 8-bit data items (byte addressing mode) or 32-bit data items (which is the default word addressing mode). Selecting byte addressing mode will also set the macro `__TS_BYTE_ADDRESS` to a value of 1. Use the `-char-size-any` switch to avoid any linking error messages if both 8-bit and 32-bit modules are used in building a project.

-const-read-write

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

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

-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.

Compiler Command-Line Interface

-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,

```
ccts -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`.

-default-branch-{np|p}

The `-default-branch-{np|p}` switch instructs the assembler and linker to set the branch behavior to be predictable or non-predictable. The default is the predicted condition.

-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.

-double-size-any

The `-double-size-any` switch directs the compiler to mark the resulting object file in such a way that it can link against objects marked with any double size (32 bits and 64 bits).

-double-size-{32 | 64}

The `-double-size-32` (double is 32 bits) and `-double-size-64` (double is 64 bits) switches select the storage format that the compiler uses for type `double`. The `-double-size-32` is the default mode.

The C/C++ type `double` poses a special problem for the compiler. The C and C++ languages default to `double` for floating-point constants and many floating-point calculations. If `double` has the customary size of 64 bits, many programs will inadvertently use slow speed emulated 64-bit floating-point arithmetic, even when variables are declared consistently as `float`. To avoid this problem, `cts` provides a mode in which `double` is the same size as `float`. This mode is enabled with the `-double-size-32` switch and is the default mode.

Representing a `double` using 32 bits gives good performance and provides enough precision for most DSP applications. This, however, does not fully conform to the C and C++ standards. The standards require that `double` maintains 10 digits of precision, which requires 64 bits of storage. The `-double-size-64` switch sets the size of `double` to 64 bits for full standard conformance.

With `-double-size-32`, a `double` is stored in 32-bit IEEE single-precision format and is operated on using fast hardware floating-point instructions. Standard math functions, such as `sin`, also operate on 32-bit values. This mode is the default and is recommended for most programs. Calculations that need higher precision can be done with the `long double` type, which is always 64 bits.

Compiler Command-Line Interface

With `-double-size-64`, a `double` is stored in 64-bit IEEE single precision format and is operated on using slow floating-point emulation software. Standard math functions, such as `sin`, also operate on 64-bit values and are similarly slow. This mode is recommended only for porting code that requires that `double` have more than 32 bits of precision.

The `-double-size-32` switch defines the `__DOUBLES_ARE_FLOATS__` preprocessor macro, while the `-double-size-64` switch undefines the `__DOUBLES_ARE_FLOATS__` preprocessor macro.

 You can invoke this switch with the **Double size** radio buttons located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category, DSP-specific selection.

-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 `-o`. Note that the `-C` switch can only be run 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.

-ED

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

-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 1-29](#)) can be used to disallow support for the additional keywords. [Table 1-7 on page 1-68](#) provides a list and a brief description of keyword extensions.

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

The `-flags` (command-line input) switch directs the compiler to pass command-line switches to the other build tools, such as:

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

Compiler Command-Line Interface

-force-circbuf

The `-force-circbuf` (circular buffer) switch instructs the compiler to make use of circular buffer facilities, even if the compiler cannot verify that the circular index or pointer is always within the range of the buffer. Without this switch, the compiler's default behavior is to be conservative, and not use circular buffers unless it can verify that the circular index or pointer is always within the circular buffer range. See [“Circular Buffer Built-In Functions” on page 1-112](#).

-fp-associative

The `-fp-associative` switch directs the compiler to treat floating-point multiplication and addition as an associative.

-full-version

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

-g

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

Compiler Command-Line Interface

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

1. The directory containing the current input file (the primary source file or the file containing the `#include`)
2. Any directories specified with the `-I` switch in the order they are listed on the command line
3. Any directories on the standard list:
`<VDSPP++ install dir>/.../include`



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

`-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.

-ipa

The `-ipa` (interprocedural analysis) switch directs the compiler to turn on the interprocedural analysis option. This option enables optimization across the entire program, including source files that are compiled separately. Using `-ipa` implicitly enables the `-O` switch. For more information, see [“Interprocedural Analysis” on page 1-62](#).



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

-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 `.dll` extension. For example, the `-lc` switch directs the linker to search in the library named “c” for functions. This library resides in a file named `libc.dll`.

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 suitable for the make utility, describing the dependencies of the main program file. The format of the make rule output by the preprocessor is:

Compiler Command-Line Interface

object-file: include-file ...

-MD

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

-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.

-Mo filename

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

-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.xml`. The `.xml` extension is added where necessary.

-mem

The `-mem` (enable memory initialization) switch directs the compiler to invoke the memory initializer tool after linking the executable. The Mem-Init tool can be controlled through the `-flags-mem` switch ([on page 1-29](#)).

-no-align-branch-lines

The `-no-align-branch-lines` (disable alignment of predicted-taken branch lines) switch directs the compiler not to align all instruction lines containing a predicted branch to a quad-word boundary.

-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](#).

-no-annotate

The `-no-annotate` (disable assembly annotations) directs the compiler not to annotate assembly files. The default behaviour is that whenever optimizations are enabled all assembly files generated by the compiler are annotated with information on the performance of the generated assembly. See “[Assembly Optimizer Annotations](#)” [on page 2-51](#) for more details on this feature.

Compiler Command-Line Interface

-no-bss

The `-no-bss` switch causes the compiler to keep zero-initialized and non-zero-initialized data in the same data section, rather than separating zero-initialized data into a different, BSS-style section. See also the `-bss` switch [on page 1-24](#).

-no-builtin

The `-no-builtin` (no built-in functions) switch directs the compiler to use generic implementations in preference to intrinsic functions for certain language extensions. For a list of the extensions that use single machine instructions when `-no-builtin` is not used, see “[Math Ininsics](#)” [on page 1-114](#). This switch also predefines the `__NO_BUILTIN` preprocessor macro.



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

-no-circbuf

The `-no-circbuf` (no circular buffer) switch disables the automatic generation of circular buffer code by the compiler. Uses of the `circindex()` and `circptr()` functions (that is, explicit circular buffer operations) will not be affected.

-no-const-strings

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

-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-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 1-37](#)) can be used to explicitly request support for the additional keywords.

-no-fp-associative

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

-no-mem

The `-no-mem` (disable memory initialization) switch directs the compiler not to run the `MemInit` memory initializer. Note that if you use `-no-mem`, the compiler does not initialize globals and statics.

-no-saturation

The `-no-saturation` switch directs the compiler not to introduce faster operations in cases where, if the expression overflowed, the faster operation would saturate, when the original operation would have wrapped the result.

Compiler Command-Line Interface

-no-std-ass

The `-no-std-ass` (disable standard assertions) switch prevents the compiler from defining the standard assertions. See the `-A` switch (on page 1-22) for the list of standard assertions.

-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.

-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.

-no-threads

The `-no-threads` (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.

-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 `ccts` compiler.

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

-O[0|1]

The `-O` (enable optimizations) switch directs the compiler to produce code that is optimized for performance. Optimizations are not enabled by default for the `ccts` compiler. The switch setting `-O` or `-O1` turns optimization on, while setting `-O0` turns off all optimizations.

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

-Oa

The `-Oa` (automatic function inlining) switch enables the inline expansion of C/C++ functions, which are not necessarily declared inline in the source code. The amount of auto-inlining the compiler performs is controlled using the `-Ov num` (optimize for speed versus size) switch ([on page 1-40](#)). The `-Ov100` setting attempts to optimize purely for speed (and therefore inline a greater number of functions) whereas the `-Ov0` setting attempts to optimize purely for space (and therefore inline fewer functions).

-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 Command-Line Interface

compiler also uses a function to save and restore preserved registers for a function instead of generating the more cycle-efficient inline default versions.

-Ov num

The `-Ov num` (optimize for speed versus size) switch directs the compiler to produce code that is optimized for speed versus size. The 'num' should be an integer between 0 (purely size) and 100 (purely speed).

-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.

-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 | mem } directory

The `-path-{asm|compiler|def|lib|link}` directory (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, linker, and

memory initializer. Use this switch when you wish to override the normal version of one or more of the tools. The `-path-{}` 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.

Compiler Command-Line Interface

-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 nonconforming 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.

-pguide

The `-pguide` (PGO) switch causes the compiler to add instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization.

-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 processor

The `-proc` processor (target processor) switch specifies that the compiler should produce code suitable for the specified processor. Refer to “[Supported Processors](#)” for the list of supported TigerSHARC processors.

For example,

```
ccts -proc ADSP-TS201 -o bin\p1.doj p1.asm
```



If no target is specified with the `-proc` switch, the system uses the ADSP-TS101 setting as a default.

If the processor identifier is unknown to the compiler, it attempts to read required switches for code generation from the file `<processor>.ini`. The assembler searches for the `.ini` file in the VisualDSP ++ System folder. For custom processors, the compiler searches the section “`proc`” in the `<processor>.ini` for key ‘`architecture`’. The custom processor must be based on an architecture key that is one of the known processors. For example, `-proc Customxxx` searches the `Customxxx.ini` file.

Compiler Command-Line Interface

When compiling with the `-proc` switch, the appropriate processor macro is defined as 1.



See also “[-si-revision version](#)” on page 1-45 for more information on silicon revision of the specified processor.

-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.

-S

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



You can invoke this switch by selecting the **Stop after: Compiler** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category 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 1-8](#) for a list of intermediate files.

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

-show

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

-si-revision version

The `-si-revision version` (silicon revision) defines the silicon revision for a particular processor, which is the required target for the build. The compiler will use this information when dealing with issues which are specific to a particular silicon revision of a processor.

The supported revisions for ADSP-TS101 processors are “0.0”, “0.1”, “0.2”, and “none”. The supported revisions for ADSP-TS20x processors are “0.0”, “0.1”, “1.0”, and “none”.

The parameter “*version*” represents a silicon revision of the processor specified by the `-proc` switch ([on page 1-43](#)).

Compiler Command-Line Interface

For example, `ccts -proc ADSP-TS201 -si-revision 0.1`

If “none” is used, these issues will be ignored. If the `-si-revision` option is not used, the compiler assumes the latest silicon revision known to it. For “0.0”, the compiler sets the `__SILICON_REVISION__` macro to 0x0. The compiler does not set the `__SILICON_REVISION__` macro for “none”.

If revision is set which the compiler does not know about, the compiler assumes the latest known revision. The `__SILICON_REVISION__` macro is then set using two hexadecimal digits each for the major and minor numbers in the revision. For example, 1.0 becomes 0x100 and 10.21 becomes 0xa15. The compiler passes the appropriate `-si-revision` switch setting when invoking another VisualDSP++ tool, for example, when the compiler driver invokes assembler and linker.

-signed-bitfield

The `-signed-bitfield` (make plain bitfields signed) switch directs the compiler to make bitfields which have not been declared with an explicit signed or unsigned keyword to be signed. This switch does not effect plain one-bit bitfields which are always unsigned. This is the default mode. See also the `-unsigned-bitfield` switch ([on page 1-48](#)).

-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:

Macro	Description
<code>__HOSTNAME__</code>	The name of the host machine
<code>__MACHINE__</code>	The 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-bitfield

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

For example, given the declaration

```
struct {  
    int a:2;
```

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

the bitfield values are:

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

See also the `-signed-bitfields` switch ([on page 1-46](#)).

-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

Compiler Command-Line Interface

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.

-vcse-add <filename>

If the **-vcse-enforce** switch has been supplied, the **-vcse-add <filename>** switch directs the compiler driver to pass **<filename>** to the **vcse_enforce** utility as the argument for the **-add** switch.

-vcse-cname <component name>

If the **-vcse-enforce** switch has been supplied, the **-vcse-cname** switch directs the compiler driver to pass **<component name>** to the **vcse_enforce** utility as the argument for the **-cname** switch.

-vcse-enforce

If the library builder (**-build-lib**) is invoked, the **-vcse-enforce** switch directs the compiler driver to invoke the **vcse_enforce** utility to process the generated library.

-vcse-names <filename>

If the **-vcse-enforce** switch has been supplied, the **-vcse-names** switch directs the compiler driver to pass **<filename>** to the **vcse_enforce** utility as the argument for the **-names** switch.

-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.

-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.

Compiler Command-Line Interface

-Wterse

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

-w

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



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

-warn-protos

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

-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` (user options) switch directs the compiler to pass the user options (but not the input *filenames*) to the main driver via a temporary file which can help if the resulting main driver command line is 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
```

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++.

-anach

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

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

- `overload` is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array may be specified in an array delete operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name may be omitted in a base class initializer if there is only one immediate base class.
- Assignment to `this` in constructors and destructors is allowed.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as an un-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.

- A reference to a `non-const` type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a `non-const` class type may be initialized from an `rvalue` of the `class` type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were proto-typed. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following statements declare the overload of two functions named `f`.

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

-eh

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

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

Object files created with exceptions enabled may be linked with objects created without, but exceptions can only be thrown from and caught, and cleanup code executed in modules compiled with `-eh`.

Compiler Command-Line Interface

-no-anach

The `-no-anach` (disable C++ anachronisms) switch directs the compiler to disallow some old C++ language features that are prohibited by the C++ standard. See the `-anach` switch ([on page 1-54](#)) for a full description of these features.

-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-eh

The `-no-eh` (disable exception handling) directs the compiler to disallow ANSI/ISO C++ exception handling. This is the default mode. See the `-eh` switch ([on page 1-55](#)) for more information.

-no-rtti

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

-rtti

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

Data Types and Data Type Sizes

The sizes of intrinsic C/C++ data types are selected by Analog Devices so that normal C/C++ programs execute with hardware-native data types and, therefore, at high speed.

Table 1-6 shows the size used for each of the intrinsic C/C++ data types when in (default) word addressing mode. For details on data type sizes when using byte addressing mode, see [“Byte Addressing Mode” on page 1-70](#). For information about the `fract` data type, refer to [“C++ Fractional Type Support” on page 1-168](#).

Table 1-6. Default Data Type Sizes for TigerSHARC Processors

Type	Word Size (Bits)	Result of sizeof operator
char, signed char, unsigned char	32	1
short, unsigned short	32	1
int, unsigned int	32	1
long, unsigned long	32	1
pointer	32	1
float	32	1
fract	32 (written with “r” suffix) (C++ mode only)	1
long long int, signed long long int, unsigned long long int	64	2
double ¹	32 or 64, float	1 or 2
long double	64, float	2

- ¹ The double size is 1 when compiled using the `-double-size-32` switch (default), and the double size is 2 when compiled using the `-double-size-64` switch (see [on page 1-27](#)).

Compiler Command-Line Interface

When compiling in word addressing mode, the compiler does not support data sizes smaller than a single word location for a processor. For the ADSP-TS101 and ADSP-TS201/202/203 processors, this means that both `short` and `char` have the same size as `int`, unless byte addressing mode is selected. Although 32-bit `chars` are unusual, they do conform to the standard.

Integer Data Types

On any platform, the basic type `int` will be the native word size—on TigerSHARC processors, it is 32 bits. Many library functions are available for both 32-bit and 64-bit integers. These functions provide support for the C/C++ data types `int` (or `long int`) and `long long int`. Pointers are the same size as `int` data types. The `long long int` data type provides 64-bit integer.

Floating-Point Data Types

On TigerSHARC processors, the `long` data type is 32 bits, the `float` data type is 32 bits, and `double` is option-selectable for 32-bit or 64-bit data. The C/C++ language tends to default to `double` for constants and for many floating-point calculations. In general, `double` word data types run more slowly than 32-bit data types because they rely largely on software-emulated arithmetic.

Type `double` poses a special problem. 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, Analog provides the “`-double-size-{32 | 64}`” switch (on page 1-27) that allows you to set the size of `double` to either 32 bits (default) or 64 bits. The 32-bit setting gives good performance and should be acceptable for most DSP programming. However, it does not conform fully to the ANSI C standard.

For a larger floating-point type, `long double` provides 64-bit floating-point arithmetic.

For either size of `double`, the standard `#include` files automatically redefine the math library interfaces so that functions, such as `sin`, can be directly called with the proper size operands.

Access to 64-bit floating-point arithmetic and libraries is always provided via `long double`.

Therefore,

```
float sinf (float);           /* 32-bit */
double sin (double);         /* 32 or 64-bit */
long double sind (long double); /* 64-bit */
```

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

Data Type Alignment

The TigerSHARC run-time model imposes some restrictions on data type alignment that are more stringent than the standard C/C++ alignment restrictions. Refer to “[Data Alignment Pragmas](#)” on page 1-150 for more information.

By default, non aggregate data objects are aligned on word boundaries (32 bits) for data types up to and including `long int`. The `long long int` and `long double` (and `double` if `double-size-64` is specified on the command line) data types are aligned on double-word boundaries (64 bits). The extended type `__builtin_quad` is aligned on a four-word boundary (128 bits).

Aggregate types have special rules:

- Top level arrays are aligned on 4-word boundaries; these are arrays that are not part of any other aggregate type. Statics and automatic arrays are aligned in this way to allow the option of using the TigerSHARC processor’s quad access instructions.

- `struct/union/class` objects are aligned to the maximal alignment of the members and have padding at the end of the structure to maintain alignment in arrays.

Optimization Control

The general aim of compiler optimizations is to generate correct code that executes fast and is small in size. Not all optimizations are suitable for every application or possible all the time so the compiler optimizer has a number of configurations, or optimization levels, which can be applied when suitable. Each of these levels is enabled by one or more compiler switches (and VisualDSP++ project options) or pragmas.

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

- **Debug**
The compiler produces debug information to ensure that the object code matches the appropriate source code line. See “-g” on [page 1-30](#) for more information.
- **Default**
The compiler does not perform any optimization by default when none of the compiler optimization switches are used (or enabled in VisualDSP++ project options). Default optimization level can be enabled using the `optimize_off` pragma ([on page 1-162](#)).
- **Procedural Optimizations**
The compiler performs advanced, aggressive optimization on each procedure in the file being compiled. The optimizations can be directed to favor optimizations for speed (`-O1` or `O`) or space (`-Os`) or a factor between speed and space (`-Ov`). If debugging is also requested, the optimization is given priority so the debugging func-

tionality may be limited. See “-O[0|1]” on page 1-39, “-Os” on page 1-39, and “-Ov num” on page 1-40. Procedural optimizations for speed and space (-O and -Os) can be enabled in C/C++ source using the pragma `optimize_{for_speed|for_space}` (for more information on optimization pragmas, see on page 1-162).

- **Profile-Guided Optimizations (PGO)**

The compiler performs advanced aggressive optimizations using profiler statistics generated from running the application using representative training data. PGO can be used in conjunction with IPA and Automatic Inlining. See “-pguide” on page 1-42 for more information.

The most common scenario in collecting PGO data will be to setup one or more simple *File to Device* streams where the *File* is a standard ASCII stream input file and the *Device* is any stream device supported by the simulator target such as memory and peripherals. The PGO process can be broken down into the execution of one or more "Data Sets" where a Data Set is the association of zero or more input streams with one and only one .pgo output file. The user can create, edit and delete the Data Sets through the IDDE and then 'Run' the Data Sets with the click of one button to produce an optimized application. The PGO operation is handled via a new PGO sub-menu added to the top-level **Tools** menu:

Tools -> PGO -> Manage Data Sets. For more information on PGO, see the *VisualDSP++ 3.5 Getting Started Guide for 32-Bit Processors* and online Help.

- **Automatic Inlining**

The compiler automatically inlines C/C++ functions which are not necessarily declared as inline in the source code. It does this when it has determined that doing so will reduce execution time. How aggressively the compiler performs automatic inlining is controlled using the -Ov switch. Automatic inlining is enabled using the -Oa

switch and additionally enables procedural optimizations (-O). See [“-Oa” on page 1-39](#), [“-Ov num” on page 1-40](#), and [“-O\[0|1\]” on page 1-39](#) for more information.



When remarks are enabled, the compiler will produce a remark to indicate each function that is inlined.

- **Interprocedural Optimizations**

The compiler performs advanced, aggressive optimization over the whole program, in addition to the per-file optimizations in procedural optimization. The *interprocedural analysis* (IPA) is enabled using the -ipa switch and additionally enables procedural optimizations (-O). See [“Interprocedural Analysis” on page 1-62](#), [“-ipa” on page 1-33](#) and [“-O\[0|1\]” on page 1-39](#) for more information.

The compiler optimizer attempts to vectorize loops when it is safe to do so. When IPA is used it can identify additional safe candidates for vectorization which might not be classified as safe at a Procedural Optimization level. Additionally, there may be other loops that are known to be safe candidates for vectorization which can be identified to the compiler with use of various pragmas (see [“Loop Optimization Pragmas” on page 1-154](#)).

Using the various compiler optimization levels is an excellent way of improving application performance. However consideration should be given to how applications are written so that compiler optimizations are given the best opportunity to be productive. These issues are the topic of Chapter 2, [“Achieving Optimal Performance from C/C++ Source Code”](#).

Interprocedural Analysis

The ccts compiler has a capability called *interprocedural analysis* (IPA), a mechanism that allows the compiler to optimize across translation units instead of within just one translation unit. This capability effectively allows the compiler to see all of the source files that are used in a final link at compilation time and make use of that information when optimizing.

Interprocedural analysis is enabled by selecting the **Interprocedural Analysis** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category, or by specifying the `-ipa` command-line switch (see [“-ipa” on page 1-33](#)).

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

 When interprocedural analysis is enabled, the pragmas controlling optimization are disabled (see [“General Optimization Pragmas” on page 1-162](#)).

Use of the `-ipa` switch causes additional files to be generated along with the object file produced by the compiler. These files have `.ipa` and `.opa` filename extensions and should not be deleted manually unless the associated object file is also deleted.

All of the `-ipa` optimizations are invoked after the initial link, whereupon a special program called the prelinker reinvokes the compiler to perform the new optimizations.

Because a file may be recompiled by the prelinker, you cannot use the `-S` option to see the final optimized assembler file when `-ipa` is enabled. Instead, you must use the `-save-temps` switch ([on page 1-45](#)), so that the full compile/link cycle can be performed first.

Interaction with Libraries

When IPA is enabled, the compiler examines all of the source files to build up usage information about all of the function and data items. It then uses that information to make additional optimizations across all of the source files. One of these optimizations is to remove functions that are never called. This optimization can significantly reduce the overall size of the final executable.

Compiler Command-Line Interface

Because IPA operates only during the final link, the `-ipa` switch has no benefit when initially compiling source files to object format for inclusion in a library. Although IPA generates usage information for potential additional optimizations at the final link stage, as normal, neither the usage information nor the module's source file are available when the linker includes a module from a library. Each library module has been compiled to the normal `-O` optimization level, but the prelinker cannot access the previously-generated additional usage information for an object in a library. Therefore, IPA cannot exploit the additional information associated with a library module.

If a library module has to make calls to a function in a user module in the program, IPA must be told that this call may occur. The reason IPA needs to know about the call is because IPA examines all the visible calls to the function and determines how best to optimize it based on that information. However, it cannot “see” the calls to the function from the library because the library code has no associated usage information to show that it uses the function.

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

```
int delete_me(int x) {
    return x-2;
}

#pragma retain_name("keep_me")
int keep_me(int y) {
    return y+2;
}

int main(void) {
    return 0;
}
```

When this program is compiled and linked with the “`-ipa`” switch, IPA can see that there are no calls to `delete_me()` in any of the source files (one source file, in this case); therefore, IPA deletes it since it is unneces-

sary. IPA does not delete `keep_me()` because the `retain_name` pragma tells IPA that there are uses of the function not visible to IPA. No pragma is necessary for `main()` because IPA knows this is the entry-point to the program.

IPA assumes that it can see all calls to a function and makes use of its knowledge of the parameters being passed to a function to effectively tailor the code generated for a function. If there are calls on a function from an object module in a library, then IPA will not have access to the information for that invocation of the function; this may cause it to incorrectly optimize the generated code.

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:

- [“Byte Addressing Mode” on page 1-70](#)
- [“Inline Function Support Keyword \(inline\)” on page 1-74](#)
- [“Inline Assembly Language Support Keyword \(asm\)” on page 1-75](#)
- [“64-Bit Integer Support \(long long\)” on page 1-87](#)
- [“Quad Word Support” on page 1-88](#)
- [“Memory Support Keywords \(pm dm\)” on page 1-88](#)
- [“__regclass Construct” on page 1-90](#)
- [“Bank Type Qualifier” on page 1-91](#)
- [“Placement Support Keyword \(section\)” on page 1-92](#)
- [“Boolean Type Support Keywords” on page 1-93](#)
- [“Pointer Class Support Keyword \(restrict\)” on page 1-93](#)
- [“Variable Length Array Support” on page 1-94](#)
- [“Non-Constant Aggregate Initializer Support” on page 1-95](#)
- [“Indexed Initializer Support” on page 1-96](#)
- [“Compiler Built-In Functions” on page 1-98](#)
- [“Pragmas” on page 1-148](#)
- [“Preprocessor Generated Warnings” on page 1-167](#)

- [“C++ Style Comments” on page 1-167](#)
- [“C++ Fractional Type Support” on page 1-168](#)
- [“GCC Compatibility Extensions” on page 1-171](#)

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 1-29](#) 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 1-7](#) provides a list and a brief description of keyword extensions. [Table 1-8](#) 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.

C/C++ Compiler Language Extensions

Table 1-7. Keyword Extensions

Keyword extensions	Description
Byte Addressing	Use the “-char-size-{8 32}” switch to select the byte addressing mode. For more information, see “Byte Addressing Mode” on page 1-70.
inline (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 1-74.
asm()	Directs the compiler to code TigerSHARC assembly language instructions within a C or C++ function. For more information, see “Inline Assembly Language Support Keyword (asm)” on page 1-75.
long long [int]	Provides 64-bit integer support, both signed and unsigned. For more information, see “64-Bit Integer Support (long long)” on page 1-87.
dm	Specifies the location of a static or global variable or qualifies a pointer declaration “*” as referring to Data Memory. For more information, see “Memory Support Keywords (pm dm)” on page 1-88.
pm	Specifies the location of a static or global variable or qualifies a pointer declaration “*” as referring to Program Memory. For more information, see “Memory Support Keywords (pm dm)” on page 1-88.
bank(“string”)	Specifies the name of the memory introduced by the user. For more information, see “Bank Type Qualifier” on page 1-91.
section(“string”)	Specifies the section in which an object or function is placed. For more information, see “Placement Support Keyword (section)” on page 1-92.
__regclass(unit)	Gives the compiler hints about which TigerSHARC unit to use for a particular calculation. For more information, see “__regclass Construct” on page 1-90.
bool, true, false	A Boolean type. For more information, see “Boolean Type Support Keywords” on page 1-93.
restrict keyword	Specifies restricted pointer features. For more information, see “Pointer Class Support Keyword (restrict)” on page 1-93.

Table 1-8. 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 1-94.
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 1-95.
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 1-96.
Preprocessor generated warnings	Support for generating warning messages from the preprocessor. For more information, see “Preprocessor Generated Warnings” on page 1-167.
C++-style comments	Support for C++-style comments in C programs. For more information, see “C++ Style Comments” on page 1-167.
<code>fract</code> data type (C++ mode)	(Provides support for the fractional data type, fractional and saturated arithmetic. For more information, see “C++ Style Comments” on page 1-167.
Quad word support	Provides support for a quad word (128 bit) data type. For more information, see “Quad Word Support” on page 1-88.

Byte Addressing Mode

To select the byte addressing mode of operation for the TigerSHARC compiler, the compiler flag `-char-size-8` (on page 1-25) should be selected both for compilation and link stages. This will also have the effect of defining the macro `__TS_BYTE_ADDRESS` with a value of 1, both when compiling and when invoking the linker. The modified `.LDF` file will, when the `__TS_BYTE_ADDRESS` macro is so defined, link against the byte address versions of the run-time libraries.

When using byte addressing mode, it is vital that the correct C/C++ header files are included in all source files.

sizeof Operator Types and Sizes

Table 1-9 shows the sizes in bits and the value returned by the `sizeof()` operator for the fundamental types that are supported by the compiler in byte addressing mode.

Table 1-9. Byte Addressing Types and Bit Lengths

Type	Size(Bits)	sizeof(T)
char, signed char, unsigned char	8	1
short, unsigned short	16	2
int, unsigned int	32	4
long, signed long, unsigned long	32	4
long long, signed long long, unsigned long long	64	8
float	32	4
double	32	4
long double	64	8
__builtin_quad	128	16
pointers	32	4

Pointers

The pointer representation uses the low-order 30 bits to address the word and the high-order two bits to address the byte within the word. Due to the pointer implementation, the address range in byte addressing mode is 0x00000000 to 0x3FFFFFFF. As noted elsewhere, the run-time support, including memory allocation, must be either the byte addressing support or the word addressing support; therefore, there is no 32-bit pointer restriction.

The main advantage of using the high-order bits to address the bytes within the word as opposed to using the low-order bits is that all pointers that address word boundaries are compatible with existing code. This choice means there will be no performance loss when accessing 32-bit items.

A minor disadvantage with this representation is that address arithmetic will be slower than using low-order bits to address the bytes within a word when the computation might involve part-word offsets.

Alignment of Objects

Within a structure, members of the fundamental types are aligned on a multiple of their size. Structures are aligned on the strictest alignment of any of their members, but are always aligned to at least 32 bits.

As always, the size of a structure is a multiple of its alignment, so this last restriction leads to subtle differences from C implementations on Windows and UNIX[®] platforms.

For example,

```
struct A { short a; short b; short c; };  
  
sizeof(struct A)      // is 8 not 6 !!!.
```

C/C++ Compiler Language Extensions

Entire variables are aligned to at least 32 bits. They will be more strictly aligned if their type requires it. Entire array variables are aligned to 128 bits.

Initializations

Initializations that require a part-word address to be computed at compile or link time will not be supported. For example,

```
short a[20];  
  
short *p = a + 3;      /* illegal */
```

The compiler will report these initializations as errors.

Pragmas Used in Byte Addressing Mode

Support for the alignment pragma has additional considerations in byte addressing mode. The following pragma

```
#pragma align n
```

is supported, although the alignment argument *n* is in bytes rather than in words and is only allowed to specify alignment on a word boundary. For more information on this pragma, refer to [“Data Alignment Pragmas” on page 1-150](#).

Performance Issues

It is quite expensive to have unoptimized loads, stores of part-words and arithmetic involving pointers to part-words. The arithmetic requires rotating the address, performing the addition or subtraction and then rotating the result. A load of a part-word must first load the word, test the high-order bits of the address, and then extract the correct part word. A store must load the entire word, deposit the part-word into it, and then write back the whole word.

Consequently, both the performance of the generated code and the size of the generated code will depend upon how successful the compiler will be at optimizing part word load and stores and pointer arithmetic.

Wherever possible, use `integer` types instead of `short` or `char` data types to improve the quality of the generated code.

Libraries Used in Byte Addressing Mode

The run-time support libraries used in byte addressing mode are independent from those used in word addressing mode. To ensure that the correct run-time support is used, programs must include the appropriate header files. Using the header files will ensure that the byte addressing run-time support routines are invoked, as the underlying entry points for the byte addressing routines are different from those used by the word addressing compiler.

The memory management routines allocate memory in units of `sizeof(char)` although the implementation will only allocate and deallocate complete words. Requests are rounded up to the next word boundary where appropriate, so that all actual memory claimed and freed are in multiples of words.



The run-time support libraries for byte addressing mode are not compatible with the word addressing support libraries. Applications have to exclusively use either the byte-addressing support libraries or the word-addressing support libraries and can not use a combination of both.

Include Files

The header files have been modified to represent the correct sizes and bounds for the data types under the `__TS_BYTE_ADDRESS` macro which is defined by the compiler under byte-addressing mode.

Inline Function Support Keyword (inline)

The `inline` keyword directs `ccts` 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 compiler only inlines functions, even those declared using the `inline` keyword, when optimizations are enabled (using the `-O` switches, as described [on page 1-39](#)).

Inline Assembly Language Support Keyword (asm)

The compiler `asm()` construct allows you to code TigerSHARC assembly language instructions within a C or C++ function. The `asm()` construct is useful in expressing assembly language statements that cannot be expressed easily or efficiently with C constructs.

The `asm()` keyword allows you to 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 a function 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("J0 = J1 + 1;;");
```

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.

C/C++ Compiler Language Extensions

Using `asm()` constructs with operands requires some additional syntax. The construct syntax is described in:

- [“Assembly Construct Template” on page 1-76](#)
- [“Assembly Construct Operand Description” on page 1-79](#)
- [“Assembly Constructs with Multiple Instructions” on page 1-85](#)
- [“Assembly Construct Reordering and Optimization” on page 1-85](#)
- [“Assembly Constructs with Input and Output Operands” on page 1-86](#)
- [“Assembly Constructs and Flow Control” on page 1-87](#)

Assembly Construct Template

Using `asm()` constructs, you can specify the operands of the assembly instruction that employ 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 semico-

lon, and enclose the entire string within double quotes. For more information on templates containing multiple instructions, see [“Assembly Constructs with Multiple Instructions” on page 1-85](#).

- **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 1-79](#).
- **output operand**
The output operand is the name of a C or C++ variable that receives output from a corresponding operand in the assembly instruction.
- **input operand**
The input operand is a C or 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. Add a space between adjacent colon field delimiters in order to avoid a clash with the C++ "::" reserved global resolution operator.
- A comma separates operands and registers within arguments.
- The number of operands in arguments must match the number of operands in your template.
- The maximum permissible number of operands is ten (%0, %1, %2, %3, %4, %5, %6, %7, %8, and %9).



The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. The compiler does not parse the assembler instruction template, does not interpret the template, and does not verify whether the template contains valid input for the assembler.

ASM() Construct Template Example

The following example shows how to apply the `asm()` construct template to the TigerSHARC assembly language `abs` instruction:

```
{  
  int result, avar;  
  ...  
  asm (  
    "%0 = %1;;":  
    "=&k" (result);  
  );  
}
```

```
    "k" (avar));
}
```

In the previous example, note the following points:

- The template is "%0 = %1;;". The %0 is replaced with operand zero (`result`), the first operand. The %1 is replaced with operand one (`avar`).
- The output operand is the C/C++ variable: `result`. The letter `k` is the *operand constraint* for the variable. This constrains the output to an IALU K-register. The compiler generates code to copy the output from the IALU register to the variable `result`, if necessary. The "=" in `=&k` indicates that the operand is an output. The & constraint ensures that the register assigned to the output cannot be the same register as the register assigned to the input.
- The input operand is the C/C++ variable (`avar`). The letter `k` is the *operand constraint* for the variable. This constrains `x` to an IALU K-register. If `avar` is stored in different kinds of registers or in memory, the compiler generates code to copy the values into an K-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 operand constraint. The compiler needs to know what kind of registers the assembly instructions can operate on, so it can allocate the correct register type.

You convey this information with a letter in the operand constraint string which describes the class of allowable registers.



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.

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 `=`.
- 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.

Operand constraints indicate what kind of operand they describe by means of preceding symbols. The possible preceding symbols are: no symbol, `=`, `+`, `&`, `?`, and `#`.

- (no symbol)

The operand is an input. It must appear as part of the third argument to the `asm()` construct. The allocated register will be loaded with the value of the C/C++ expression before the `asm()` template is executed. Its C/C++ expression will not be modified by the `asm()`, and its value may be a constant or literal. Example: `x`

- `= symbol`

The operand is an output. It must appear as part of the second argument to the `asm()` construct. Once the `asm()` template has been executed, the value in the allocated register is stored into the location indicated by its C/C++ expression; therefore, the expression must be one that would be valid as the left-hand side of an assignment.

Example: `=x`

- `+ symbol`

The operand is both an input and an output. It must appear as part of the second argument to the `asm()` construct. The allocated register is loaded with the C/C++ expression value, the `asm()` template is executed, and then the allocated register's new value is stored back into the C/C++ expression.

Therefore, as with pure outputs, the C/C++ expression must be one that is valid on the left-hand side of an assignment.

Example: `+x`

- **? symbol**

The operand is temporary. It must appear as part of the third argument to the `asm()` construct. A register is allocated as working space for the duration of the `asm()` template execution. The register's initial value is undefined, and the register's final value is discarded. The corresponding C/C++ expression is not loaded into the register, but must be present. This expression is normally specified using a literal zero. Example: `?x`

- **& symbol**

This operand constraint may be applied to inputs and outputs. It indicates that the register allocated to the input (or output) may not be one of the registers that are allocated to the outputs (or inputs). This operand constraint is used when one or more output registers are set while one or more inputs are still to be referenced. (This situation sometimes occurs if the `asm()` template contains more than one instruction.) Example: `&x`

- **# symbol**

The operand is an input, but the register's value is clobbered by the `asm()` template execution. The compiler may make no assumptions about the register's final value. The operand must appear as part of the second argument to the `asm()` construct. Example: `"#x"`

It is also possible to claim registers directly, instead of requesting a register from a certain class using the constraint letters. You can claim the registers directly by simply naming the register in the location where the class letter would be.

For example,

```
asm("%0 += FDEP %1 BY %2;;"
    :"+XR0"(val)      /* output */
    : "X"(x), "X"(y)   /* input  */
    );
```

would load `val` into `XR0`, and load `x` and `y` into two `X` registers, execute the operation, and then store the new value from `XR0` back into `val`.

[Table 1-10](#) provides the constraint register types. The use of any letter not listed in the “Constraint” column of this table results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter. [Table 1-11](#) describes constraint operators.



Naming the registers in this way allows the `asm()` to specify several registers that must be related, such as the Base and Length registers for a circular buffer. This also allows use of register not covered by the register classes accepted by the `asm()`.

The clobber string can be any of the registers recognized by the compiler. These should all be in upper case, for example “`XR10`”, “`YMR0`”, “`XTR3`”, or “`LC0`”.

Table 1-10. Constraint Register Types

<i>Constraint</i> ¹	Register Type
x	Compute block X register
y	Compute block Y register
j	JALU register
k	KALU register

¹ Names are case-sensitive.

Table 1-11. Constraint Operators

Constraint Operator ¹	Description
<i>constraint</i>	Indicates the constraint is an input operand
<i>constraintl</i>	Indicates the constraint is long (64bit) register
<i>constraintq</i>	Indicates the constraint is quad (128-bit) register
<i>=constraint</i>	Indicates the constraint is applied to an output operand
<i>&constraint</i>	Indicates the constraint is applied to an input operand that may not be overlapped with an output operand
<i>=&constraint</i>	Indicates the constraint is applied to an output operand that may not overlap an input operand
<i>?constraint</i>	Indicates the constraint is temporary
<i>+constraint</i>	Indicates the constraint is both an input and output operand
<i>#constraint</i>	Indicates the constraint is an input operand whose value will be changed
<i>constraintb</i>	Indicates the constraint is applied to a byte operation
<i>constraints</i>	Indicates the constraint is applied to a short operation.
<i>constraintL</i>	Indicates the constraint is applied to a long operation.
<i>constraintf</i>	Indicates the constraint is applied to a float operation.

¹ Names are case-sensitive.

Note that if the *q* or *l* constraints are specified, then they must be specified before the constraints listed above. For example,

For the following inline assembler statement:

```
asm("%0 = LSHIFT %1 BY -1;;" : "=x1L" (input) : "x1" (input) );
```

where *input* and *output* are defined as `long long` types would produce a core instruction similar to

```
XLR15:14 = LSHIFT R13:12 BY -1;;
```

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:

```
/* (pseudo code) k2 = from; k3 = to; result = from + to; */
asm ("k2=%1;; \
    k3=%2;; \
    %0=k2+k3;;"
    : "=k" (result)           /* output */
    : "k" (from), "k" (to)    /* input */
    : "k2", "k3");           /* 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 GetCycleCount(counter) \
asm volatile ("%0 = CCNT0;;" : "=j"(counter) );
```

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; the compiler 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.

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

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

```
asm("[%0 +=2] = J4;;"  
    : "=j" (newptr)           // an output, given a jreg,  
                                // stored into newptr.  
    : "0" (oldptr));         // an input, given same reg as %0,  
                                // initialized from oldptr  
    )
```

Assembly Constructs and Flow Control

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

For example, the compiler is careful to adhere to the calling conventions for preserved registers when making a procedure call. If an `asm()` construct calls a procedure, the `asm()` construct must also ensure that all conventions are obeyed, or the called procedure may corrupt the state used by the function containing the `asm()` construct.

64-Bit Integer Support (long long)

In addition to the basic C/C++ data types, the `ccts` compiler provides an additional integral type that consists of 64-bit integers. This type is provided in both signed and unsigned forms. It is fully supported, including all arithmetic operations and relevant libraries.

Type Name	<code>long long [int]</code> <code>unsigned long long [int]</code>
Representation	64-bit 2's complement
Range	<code>long long int</code> : $-2^{63} \dots (2^{63})-1$ <code>unsigned long long int</code> : $0 \dots (2^{64}) - 1$

Normal library functions (as in `<stdlib.h>`) taking this type typically have an “`ll`” prefix. Formatting support (`printf`, etc.) is available via the “`ll`” modifier, as in “`%lld`”.



This extension does not introduce additional keywords, so it does not interfere with porting standard-conforming programs onto TigerSHARC processors. The extension is always available; the `-pedantic` switch does not affect it.

Quad Word Support

Certain features of TigerSHARC processors make use of 128-bit data types (a quad word). An extension to the language provides a data type of this size. Normal C/C++ arithmetic operators are not supported for the quad word type; therefore, supplied built-in functions must be used to process variables of this type. The type name is `__builtin_quad`.

Memory Support Keywords (pm dm)

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.

Memory Keyword Rules

The following rules apply to 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 (`dm`) as the default memory space for all variables. All undeclared spaces for data are Data Memory spaces.
- 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.

The following listing shows examples of memory keyword syntax.

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

__regclass Construct

The `__regclass(unit)` construct allows you to give the compiler hints about which TigerSHARC processor unit to use for a particular computation. Variables can be annotated with this construct, and the compiler then tries to perform computations involving those variables in the indicated units. The `(unit)` argument is a quoted string, naming one of the four major units: “X”, “Y”, “J”, “K”.

The `__regclass` construct may be used anywhere that the C register qualifier may be used and is subject to the same restrictions: it cannot have its address taken. Effective use of this feature requires a good understanding of the TigerSHARC architecture. Typically, you will examine the assembly code produced by the optimizing compiler.

In the absence of specific `__regclass` hints, the TigerSHARC compiler uses the following criteria:

- Floating-point calculations are done in compute block X.
- General integer calculations are done in compute block Y.
- Address calculations, including pointer arithmetic, are done in the JALU. If a multiplication is needed, it is done in Y.

A common way of achieving higher performance, particularly in tight loops, is to perform a part of the computation in a different unit. If the subcomputations assigned to different units are independent of each other, they will execute in parallel, resulting in increased throughput.

Bank Type Qualifier

This type-qualifier specifies the name of the memory introduced by the user. The new type-qualifier is `bank` and is syntactically represented as

```
bank(<string-literal>)
```

A maximum of ten different memory banks may be specified in a source compilation. Function overloading on memory banks in C++ is legal.

Example:

```
// C++ example
void func(int bank("apple") *par);
void func(int bank("banana") *par);
void other_func(int bank("pear") *par);
int bank("apple") *my_apple;
int bank("banana") *my_banana;
int bank("pear") *my_pear;
void test()
{
    func(my_apple); // always match void func(int bank("apple") *par);
    func(my_banana); // always match void func(int bank("banana") *par);
    func(my_pear); // ambiguous if memory banks are compatible
```

```
        // no match if memory banks are not compatible
other_func(my_apple); // match void other_func(int bank("pear") if
        // memory banks compatible.
        // no match if memory banks are not compatible.
other_func(my_banana); // match void other_func(int bank("pear") if
        // memory banks compatible.
        // no match if memory banks are not compatible
other_func(my_pear); // always match void other_func(int
        // bank("pear") *par);
}
```

For details on how the bank qualifier can be used, refer to [“Memory Usage” on page 2-19](#).

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, for example, they 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

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.

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, such as:

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

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

As an example, if you were to implement a routine for computation of a product of three matrices, you need to allocate a temporary matrix of the same size as input matrices. Declaring 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 ccts 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

The 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 ccts 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 `ccts` compiler.

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

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

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

/* equivalent ccts 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 `ccts` 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 & ccts C/C++ Array Initializer */
/* Standard Array Initializer */

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

/* equivalent ccts 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 */
/* ccts 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 ccts C/C++ struct initializers in the example below are equivalent.

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

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

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

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

Compiler Built-In Functions

The compiler supports intrinsic (built-in) functions that enable you to make efficient use of hardware resources. The compiler is aware of the definition of these intrinsic functions without the use of a header file. 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 '+' or '*'.

Built-in functions have names that begin with `__builtin`.



Identifiers beginning with '___' are reserved by the C standard, so these names do not conflict with user-defined identifiers.

This section describes:

- [“Using the builtins.h Header File” on page 1-99](#)
- [“Optimization Guidance Built-in Functions” on page 1-101](#)
- [“16-Bit Data Types” on page 1-101](#)
- [“32-Bit Data Types” on page 1-111](#)

- “Circular Buffer Built-In Functions” on page 1-112
- “Math Intrinsics” on page 1-114
- “Instructions Generated by Built-in Functions” on page 1-115
- “Data Alignment Buffer (DAB) Built-in Functions” on page 1-129
- “Circular Buffer Data Alignment Buffer (DAB) Built-in Functions” on page 1-131
- “Communications Logic Unit Operations” on page 1-132

These functions are specific to individual architectures; the built-in functions supported on TigerSHARC processors are described in subsections that follow.

Various system header files provide the user with definitions and access to the intrinsics. This feature may be disabled using the `-no-builtin` switch (see [on page 1-36](#)).

Using the `builtins.h` Header File

The `builtins.h` header file provides user-visible prototypes for the built-in functions, though as noted above, these are not necessary for the compiler as it already has the information about the built-in prototypes. It does, however, also provide shorthand forms of some of the more complicated built-in functions (and it is strongly recommended that these are used over their constituent built-in functions).

The `builtins.h` header file also provides C reference code for the built-in functions. The header file can thus be included in compiler for architectures other than the TigerSHARC processors and can enable compilation and execution of code that makes use of the TigerSHARC built-in functions. Examining the C reference code can provide insight into the function that a given built-in function performs.

C/C++ Compiler Language Extensions

When using a compiler other than the one provided with VisualDSP++ for TigerSHARC processors, please make sure your compiler supports inline functions as the C reference versions of the built-ins use the `inline` keyword. These implementations of the built-in functions also require support for 64-bit integers.

When using this file on a machine that does not support byte-addressing, define `__NO_BYTE_ADDRESSING__` before including this file. If the machine has 8-bit `char` types and 16-bit `short` types, then do not define this macro.

To use the reference implementation of the `XCORRS` Communication Logic Unit instruction in projects being built for the ADSP-TS101 processor, define the macro `__USE_RAW_XCORRS__` before including the `builtins.h` file.

When using the Microsoft C/C++ compiler, use the `/TP` command-line option to select C++ compilation: “`inline`” is not recognised as a keyword in C compilation mode.

To allow your compilation to ignore all of the `__builtin_sysreg_read` and `__builtin_sysreg_write` built-in functions, define `__IGNORE_SYSREG_BUILTINS__` before including this file which will allow the code to compile. To allow your compilation to ignore the `__builtin_idle` built-in functions, define `__IGNORE_IDLE_BUILTINS__` before including this file.

To use the raw versions of the built-in functions even when using the compiler provided with VisualDSP++ for TigerSHARC processors, define `__USE_RAW_BUILTINS__` before including this file.

Optimization Guidance Built-in Functions

This section describes the optimization guidance built-in functions.

```
void __builtin_aligned(const void *p, int align);
```

The `__builtin_aligned` intrinsic is an assertion that the first parameter points to an argument that is aligned on a multiple of the second argument. The second argument is in units of `char` size, that is 8-bit bytes in byte-addressing mode and 32-bit words in word-addressing mode. Knowing alignment helps the optimiser to combine loads and stores from successive iterations of a loop. Ideally all data processed in the first iteration of the loop should be aligned on a 4 word boundary.

An example in word-addressing mode where the parameter “a” points to quad-word aligned data would be as follows:

```
void fn(int *a) {
    __builtin_aligned(a, 4);
    ....
}
```

16-Bit Data Types

The functions and operators described in this section perform arithmetic on the `int2x16`, and `int4x16` data types.

- The `int2x16` data type consists of two 16-bit integers packed into a 32-bit structure.
- The `int4x16` data type consists of four 16-bit integers packed into a 64-bit item.

In the following table, the notation `{A,B}` represents an `int2x16` containing the two 16-bit values A and B. The TigerSHARC processors have a little endian architecture, so the “first” element is stored in the low order

bits. A is considered to be the “first” element of {A,B}, so that the representation in a 32-bit register (`int2x16` data type) has the following structure.

31	16	15	0
Value B		Value A	
Bit 15 Bit 0		Bit 15 Bit 0	

Similarly, {A,B,C,D} represent an `int4x16` containing the 16-bit values A, B, C, and D, with A in the lowest order bits, B next, followed by C, and D in the highest order bits. (This means that C and D will be stored in a register or memory location numbered one higher than A and B.)

Most operations on these types are element-wise. For example, the add operation for `int2x16` takes the first 16-bit quantity from each argument and adds them together to produce the first 16-bit quantity in the result. Likewise, the second 16-bit elements are added to produce the second element of the result. For example, the `int2x16` add operation on arguments {A,B} and {C,D} produces a result {A+C,B+D}.

A few operations which are not element wise are also supported. The side-wys sum operation adds up all of the 16-bit elements in its single argument. A variety of packing and unpacking operations for accessing the individual elements of a packed value are also available.

These operations are very efficient for the ADSP-TS101 processors because they are directly supported in hardware. Most are implemented as a single machine instruction; a few which can take multiple machine instructions are provided for convenience. Note that some operations are defined in terms of a `2x32` data type. This type and its operations are described in [“32-Bit Data Types” on page 1-111](#).

Packed 16-bit Integer Support Using C

The C language version used at Analog Devices lets you write your own program to operate on 16-bit packed data. To use the `int2x16` and `int4x16` types and operators in your Analog Devices C program, add a `#include <i16.h>` directive.

Constructors (`int2x16` values)

The following functions create `int2x16` values.

Synopsis:

```
int2x16 compact_to_i2x16_from_i32 (int lo, int hi);
```

Description: Constructs an `int2x16` type by compacting two 32-bit integer values. The high-order 16 bits of each original 32-bit value are lost.

Algorithm:

```
compact(A_32, B_32) => {A_16,B_16}
```

Synopsis:

```
int2x16 compact_to_i2x16 (int2x32 val);
```

Description: Construct an `int2x16` type by compacting the two halves of a single `int2x32`. The high-order 16 bits of each of the 32-bit values are lost.

Algorithm:

```
compact({A_32,B_32}) => {A_16,B_16}
```

Extractors and Expanders (`int2x16` values)

The following functions extract the latest or most significant 16-bit value from an `int2x16`.

Synopsis:

```
int expand_low_of_i2x16 (int2x16 val);  
int expand_high_of_i2x16 (int2x16 val);
```

Description: The value returned is sign-extended to 32 bits and returned as an `int`.

Algorithm:

```
val = {A_16,B_16}:  
expand_low_of_i2x16(val) => A_32  
expand_high_of_i2x16(val) => B_32
```

See Also: `expand_i2x16_to_i2x32`

Arithmetic Operators

The following binary operators are defined on `int2x16`:

Synopsis:

```
int2x16 add_i2x16 (int2x16 a, int2x16 b);  
int2x16 sub_i2x16 (int2x16 a, int2x16 b);  
int2x16 mult_i2x16 (int2x16 a, int2x16 b);
```

Description: Perform element-wise arithmetic on `int2x16` values. The TigerSHARC processors do not have a `2x16` multiplication instruction; this multiply operation uses the `4x16` multiply and discards half of the result. It is still accomplished in one instruction.

Algorithm:

```
Addition:      {A,B} + {X,Y} => {A+X, B+Y}  
Subtraction:    {A,B} - {X,Y} => {A-X, B-Y}  
Multiplication: {A,B} * {X,Y} => {A*X, B*Y}
```

Unary minus is not yet supported on `int2x16`.

Bitwise Operators (int2x16 values)

The following bitwise logical operators are defined on `int2x16`:

Synopsis:

```
int2x16 a, b;
a ^ b
a & b
a | b
a ^= b
a &= b
a |= b
~a
```

Description: Perform logical operations on `int2x16` values. Note that the traditional `infix` operators are available here.

Algorithm:

Xor:	$\{A, B\} \wedge \{X, Y\} \Rightarrow \{A \wedge X, B \wedge Y\}$
And:	$\{A, B\} \& \{X, Y\} \Rightarrow \{A \& X, B \& Y\}$
Or:	$\{A, B\} \mid \{X, Y\} \Rightarrow \{A \mid X, B \mid Y\}$
Complement:	$\{A, B\} \Rightarrow \{\sim A, \sim B\}$

Comparison Operators (int2x16 values)

The following operators are used to compare `int2x16` values.

Synopsis:

```
int2x16 a, b;
a == b;
a != b;
```

Description: The `==` operator returns true if both elements are equal. The `!=` operator returns true unless both values are equal. Other comparison operators (`<`, `<=`, `>=`, `>`) are not supported.

C/C++ Compiler Language Extensions

Algorithm:

Equal: $\{A,B\} == \{C,D\} \Rightarrow (A == B \ \&\& \ C == D)$

Not Equal: $\{A,B\} != \{C,D\} \Rightarrow (A != B \ || \ C != D)$

Sideways Sum (int2x16 values)

This function adds together the two values in an `int2x16`.

Synopsis:

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

Description: This operation produces a single 32-bit integer result.

Algorithm:

```
sum({A,B}) => A+B
```

Example (int2x16 values)

This example shows some of the `int2x16` operations, including conversion to and from normal ints. Note that it is often helpful in debugging to print the packed values in hex, rather than decimal, as the two halves can be seen more easily.

```
#include <stdio.h>           // for printing at end
#include <i16.h>
void i2x16_example() {
    int u, v, s;
    int
        x = 3,
        y = 5,
        z = 7,
        w = 9;
    int2x16 aa, bb, cc, dd;
        // compact integers into packed 16 bit form
    aa = compact_to_i2x16_from_i32(x, y);
        // aa = {3, 5} or 0x00050003
    bb = compact_to_i2x16_from_i32(z, w);
        // bb = {7, 9} or 0x00090007
```

```

        // construct a packed constant
cc = compact_to_i2x16_from_i32(1, -1);
    // cc = {1, -1} or 0xffff0001
    // a little arithmetic
dd = mult_i2x16 (add_i2x16 (aa, bb), cc);
    // dd = (aa + bb) * cc;
    // dd = {10, -14} or 0xffff2000a
    // sideways sum
s = sum_i2x16(dd);                //s = -4 or 0xffffffffc
                                   // extract components
u = expand_low_of_i2x16(dd);      // low-order part
v = expand_high_of_i2x16(dd);     // high-order part
printf("results: s = %d; dd = %8x, u = %d, v = %d\n", s, dd, u, v);
}

Prints:
results: s = -4; dd = fff2000a, u = 10, v = -14

```

Constructors (int4x16 values)

The following functions create int4x16 values

Synopsis:

```
int4x16 compact_to_i4x16_from_i32 (int llo, int lhi,
                                   int hlo, int hhi);
```

Description: Construct an int4x16 by compacting four 32-bit integer values. The high-order 16 bits of each original 32-bit value are lost.

Algorithm:

```
compact(A_32, B_32, C_32, D_32) => {A_16,B_16,C_16,D_16}
```

Synopsis:

```
int4x16 compose_i4x16_from_i32 (int2x16 low, int2x16 high);
```

Description: Construct an int4x16 from two int2x16s.

Algorithm:

```
compose ({A, B}, {C, D}) => {A,B,C,D}
```

Extractors (2x16 from a 4x16 value)

This function extracts a 2x16 from a 4x16

Synopsis:

```
int2x16 extract_low_of_i4x16(int4x16 val);  
int2x16 extract_high_of_i4x16(int4x16 val);
```

Description: Extracts a int2x16 from a int4x16. Often it is convenient to use the various expand operators to break the int4x16 apart in steps, holding onto the intermediate forms, as shown in the following listing.

Given:

```
int4x16 val = {A_16,B_16,C_16,D_16}; int2x16 low_part =  
    extract_low_of_i4x16(val);  
int2x16 high_part = extract_high_of_i4x16(val);  
int a = expand_low_of_i2x16(low_part);  
int b = expand_high_of_i2x16(low_part);  
int c = expand_low_of_i2x16(high_part);  
int d = expand_high_of_i2x16(high_part);
```

- or -

```
int4x16 val = {A_16,B_16,C_16,D_16};  
int2x32 low_part = expand_i2x16(extract_low_of_i4x16(val));  
int2x32 high_part = expand_i2x16(extract_high_of_i4x16(val));  
int a = low_32(low_part);  
int b = high_32(low_part);  
int c = low_32(high_part);  
int d = high_32(high_part);
```

Split an int4x16 into component integer values

Algorithm:

```
extract_low ({A_16,B_16,C_16,D_16}) => {A_16,B_16}
extract_high({A_16,B_16,C_16,D_16}) => {C_16,D_16}
```

Synopsis:

```
void expand_i4x16_to_i32(int4x16 input,
    int *llo, int *lhi, int *hlo, int *hhi);
```

Description: Using a cascade of expansion and selection, this breaks a 4x16 apart into four separate integers.

Algorithm:

```
val = {A_16,B_16,C_16,D_16}:
int a, b, c, d;
expand_i4x16_to_i32 (val, &a, &b, &c, &d);
a => A_32
b => B_32
c => C_32
d => D_32
```

Arithmetic Operators (int4x16 values)

The following binary operators are defined on int4x16:

Synopsis:

```
int4x16 add_i4x16    (int4x16 a, int4x16 b);
int4x16 sub_i4x16    (int4x16 a, int4x16 b);
int4x16 mult_i4x16   (int4x16 a, int4x16 b);
```

Algorithm:

Addition: $A,B,C,D\} + \{W,X,Y,Z\} \Rightarrow \{A+W, B+X, C+Y, D+Z\}$
Subtraction: $A,B,C,D\} - \{W,X,Y,Z\} \Rightarrow \{A-W, B-X, C-Y, D-Z\}$
Multiplication: $A,B,C,D\} * \{W,X,Y,Z\} \Rightarrow \{A*W, B*X, C*Y, D*Z\}$

Description: Perform element wise arithmetic on int4x16 values. Unary minus is not yet supported on int4x16.

Sideways Sum (int4x16 values)

This function adds together the four values in an `int4x16`.

Synopsis:

```
int sum_i4x16(int4x16);
```

Description: This operation produces a single 32-bit integer result.

Algorithm:

```
sum({A,B,C,D}) => A+B+C+D
```

Example (int4x16 values)

The following shows some examples of `int4x16` operations, including conversion to and from normal `ints`.

```
#include <stdio.h>
#include <i16.h>
void i4x16_example() {
    int w, x, y, z, s;
    int
        p = 3,
        q = 5,
        r = 7,
        s = 11,
        t = 13,
        u = 17,
        v = 19,
        g = 23;
    int4x16 aaaa, bbbb, cccc, dddd;
    // compact integers into packed 16 bit form
    aaaa = compact_to_i4x16_from_i32(p, q, r, s);
    bbbb = compact_to_i4x16_from_i32(t, u, v, g);
    cccc = compact_to_i4x16_from_i32(1, -1, 3, -3);
    // construct a packed constant
    // a little arithmetic
    dddd = mult_i4x16 (add_i4x16 (aaaa, bbbb), cccc);
    // dddd = (aaaa + bbbb) * cccc;
```

```

    s = sum_i4x16(dddd);    // sideways sum
                           // extract components
    w = dddd[0];           // low-order part
    x = dddd[1];           // high-order part
    y = dddd[2];
    z = dddd[3];
    printf("results: s = %d; w = %d, x = %d, y = %d, z = %d\n",
           w, x, y, z);
}

```

Prints:

```
results: sm = -30; w = 16, x = -22, y = 78, z = -102
```

32-Bit Data Types

The `int2x32` data type is primarily used internally when converting between packed and expanded values. It often provides a useful way-station, allowing you to capture a partial expansion for later separation into smaller parts.

Constructors (`int2x32` values)

The following functions create `int2x32` values.

Synopsis:

```
int2x32 compose_64 (int lo, int hi);
```

Algorithm:

```
compose(A_32, B_32) => {A_32, B_32}
```

Description: Constructs an `int2x32` from two 32-bit integer values. The values are unchanged.

Synopsis:

```
int2x32 expand_i2x16 (int2x16 val)
```

C/C++ Compiler Language Extensions

Algorithm:

```
expand ({A_16,B_16}) => {A_32, B_32}
```

Description: Expands an `int2x16` to an `int2x32` sign extending each value to 32 bits.

Extractors (`int2x32` values)

The following functions extract the least or most significant 32-bit value from an `int2x32`. The unchanged 32-bit value is returned as an `int`.

Synopsis:

```
int low_32 (int2x32 val);  
int high_32 (int2x32 val);
```

Algorithm:

```
low(A_32, B_32) => A_32  
high(A_32, B_32) => B_32
```

Circular Buffer Built-In Functions

The C/C++ compiler provides the following two built-in functions for using the TigerSHARC circular buffer mechanisms.



If the compiler is being used in byte addressing mode, the circular buffer support will only work for types of size `int` and greater. With the current compiler version, use of `short`, `char` or `void` pointers could result in incorrect run-time behavior.

Circular Buffer Increment of an Index

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

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

The operation is:

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

Circular Buffer Increment of a Pointer

The following operation

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

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

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

The result of this operation is the updated circular buffer pointer.

The compiler also attempts to generate circular buffer increments for modulus array references, such as `array[index %nitems]`. For this to happen, the compiler must be able to determine that the starting value for `index` will be within the range `0..(nitems-1)`.

When the “[-force-circbuf](#)” switch (on [page 1-30](#)) is specified, the compiler will always treat array references of the form `"[i%n]"` as a circular buffer operation on the array.

Math Intrinsics

Each of the functions below map directly to a single machine instruction and are therefore very efficient. When using these functions, the compiler will substitute machine instructions unless you specify the command line “-no-builtin” option ([on page 1-36](#)).

```
/* Intrinsics defined in <stdlib.h> */
/* "int" versions */
int  abs  (int j);           // absolute value
int  avg  (int a, int b);    // (a+b)/2
int  clip (int a, int b);
    // bound a by b (positive and negative)
int  max  (int a, int b);    // maximum
int  min  (int a, int b);    // minimum
int  count_ones (int numb);  // number of "1" bits

/* "long int" versions */
long labs  (long j);
long lavg  (long a, long b);
long lclip (long a, long b);
long lmax  (long a, long b);
long lmin  (long a, long b);

int  lcount_ones (long numb);

/* "long long int" versions */
long long int  llabs (long long int j);
long long int  llavg  (long long int a, long long int b);
long long int  llclip (long long int a, long long int b);
long long int  llmax  (long long int a, long long int b);
long long int  llmin  (long long int a, long long int b);

int  llcount_ones (long long int numb);

/* bit-reversed addition */
int  addbitrev (int input_address, int number_of_bits);

/* Math intrinsics from <math.h> */
float  fabsf (float, float);
```

```

float   favgf (float a, float b);    // (a+b)/2
float   fclipf (float a, float b);  // bound a by b (pos & neg)
float   fmaxf (float, float);        // float maximum
float   fminf (float, float);        // float minimum
float   copysignf (float a, float b); // a with sign(b)
float   signif (float a, float b);   // a with sign(b)

double  favgd (double, double);
double  fclipd (double, double);
double  fmaxd (double, double);
double  fmind (double, double);
double  copysignd (double, double);

long double  favgd (long double, long double);
long double  fclipd (long double, long double);
long double  fmaxd (long double, long double);
long double  fmind (long double, long double);
long double  copysignd (long double, long double);

```

Instructions Generated by Built-in Functions

The `ccts` compiler supports access to machine-specific instructions via built-in functions. These functions are known to the compiler without the need to declare them (via a header file). However, for convenience, a header file is included with the other C-includes (`builtins.h`). This header file gives prototypes for all the built-in functions. This file is quite terse, but it does provide the correct call syntax for each function.

Every function name is prefixed with `__builtin_` followed by a functional name, which usually corresponds directly to the machine instruction that the built-in function generates. Some functions, such as `compose` or `extract`, may not generate code under optimization since the compiler may be able to arrange register assignments to eliminate the need for these functions.

The following are listings of the built-ins functions currently supported by the compiler and the instructions that they generate. Note that the instructions listed are the ones the compiler will use by default. As part of the optimization process, the compiler may replace instruction sequences with alternate equivalent sequences.

Many of the built-in functions map to a single machine instruction. The first section of this list comprises these builtins. Further down in the list are instructions which do not correspond directly to a machine instruction but which can be implemented simply by other means (for example, extracting the low half of a 64-bit item can be done by the compiler making use of the low half of the item and ignoring the upper half). More complex builtins are dealt with in separate sections based on their function. For details on a specific function, please refer to the Instruction Set Reference, the Programming Reference, or the Hardware Reference for the appropriate TigerSHARC target processor.

Addition and Subtraction:

```
int __builtin_add_sat(int, int);
    Rs = Rm + Rm (S);
int __builtin_sub_sat(int, int);
    Rs = Rm - Rm (S);
unsigned int __builtin_uadd_sat(unsigned int, unsigned int);
    Rs = Rm + Rm (SU);
unsigned int __builtin_usub_sat(unsigned int, unsigned int);
    Rs = Rm - Rm (SU);
int __builtin_add_4x8(int, int);
    SRs = Rm + Rn;
int __builtin_add_4x8_sat(int, int);
    SRs = Rm + Rn (S);
int __builtin_sub_4x8(int, int);
    SRs = Rm - Rn;
int __builtin_sub_4x8_sat(int, int);
    SRs = Rm - Rn (S);
int __builtin_add_2x16(int, int);
    SRs = Rm + Rn;
int __builtin_add_2x16_sat(int, int);
```

```

    SRs = Rm + Rn (S);
int __builtin_sub_2x16(int, int);
    SRs = Rm - Rn;
int __builtin_sub_2x16_sat(int, int);
    SRs = Rm - Rn (S);

unsigned int __builtin_add_u2x16(unsigned int, unsigned int);
    SRs = Rm + Rn;
unsigned int __builtin_add_u2x16_sat(unsigned int, unsigned int);
    SRs = Rm + Rn (SU);
unsigned int __builtin_sub_u2x16(unsigned int, unsigned int);
    SRs = Rm - Rn;
unsigned int __builtin_sub_u2x16_sat(unsigned int, unsigned int);
    SRs = Rm - Rn (SU);

long long int __builtin_add_8x8(long long int, long long int);
    SRsd = Rmd + Rnd;
long long int __builtin_add_8x8_sat(long long int,
                                   long long int);
    SRsd = Rmd + Rnd (S);
long long int __builtin_sub_8x8(long long int, long long int);
    SRsd = Rmd - Rnd;
long long int __builtin_sub_8x8_sat(long long int,
                                   long long int);
    SRsd = Rmd - Rnd (S);
long long int __builtin_add_4x16(long long int, long long int);
    SRsd = Rmd + Rnd;
long long int __builtin_add_4x16_sat(long long int,
                                   long long int);
    SRsd = Rmd + Rnd (S);
long long int __builtin_sub_4x16(long long int, long long int);
    SRsd = Rmd - Rnd;
long long int __builtin_sub_4x16_sat(long long int, long long int);
    SRsd = Rmd - Rnd (S);

unsigned long long int __builtin_add_u4x16_sat(
    unsigned long long int,
    unsigned long long int);
    SRsd = Rmd + Rnd (SU);

```

C/C++ Compiler Language Extensions

```
unsigned long long int __builtin_sub_u4x16_sat(
    unsigned long long int,
    unsigned long long int);

    SRsd = Rmd - Rnd (SU);
long long int __builtin_add_2x32(long long int, long long int);
    Rsd = Rmd + Rnd;
long long int __builtin_add_2x32_sat(long long int,
    long long int);

    Rsd = Rmd + Rnd (S);
long long int __builtin_sub_2x32(long long int, long long int);
    Rsd = Rmd - Rnd;
long long int __builtin_sub_2x32_sat(long long int,
    long long int);

    Rsd = Rmd - Rnd (S);
unsigned long long int __builtin_add_2x32u(
    unsigned long long int,
    unsigned long long int);

    Rsd = Rmd + Rnd;
unsigned long long int __builtin_add_u2x32_sat(
    unsigned long long int,
    unsigned long long int);

    Rsd = Rmd + Rnd (SU);
unsigned long long int __builtin_sub_2x32u(
    unsigned long long int,
    unsigned long long int);

    Rsd = Rmd - Rnd;
unsigned long long int __builtin_sub_u2x32_sat(
    unsigned long long int,
    unsigned long long int);

    Rsd = Rmd - Rnd (SU);

__builtin_quad __builtin_add_4x32(__builtin_quad,
    __builtin_quad);

    two separate Rsd = Rmd + Rnd;
__builtin_quad __builtin_add_4x32_sat(__builtin_quad,
    __builtin_quad);

    two separate Rsd = Rmd + Rnd (S);
__builtin_quad __builtin_add_u4x32_sat(__builtin_quad,
    __builtin_quad);

    two separate Rsd = Rmd + Rnd (SU);
```

```

__builtin_quad __builtin_sub_4x32(__builtin_quad,
                                   __builtin_quad);

    two separate Rsd = Rmd - Rnd;
__builtin_quad __builtin_sub_4x32_sat(__builtin_quad,
                                       __builtin_quad);

    two separate Rsd = Rmd - Rnd (S);
__builtin_quad __builtin_sub_u4x32_sat(__builtin_quad,
                                       __builtin_quad);

    two separate Rsd = Rmd - Rnd (SU);
int __builtin_addbitrev(int, int);
    Js = Jm + Jn (BR);

```

Conversion:

```

int __builtin_compact_to_fr2x16(long long int);
    SRs = COMPACT Rmd;
int __builtin_compact_to_fr2x16_sat(long long int);
    SRs = COMPACT Rmd (S);
int __builtin_compact_to_i2x16(long long int);
    SRs = COMPACT Rmd (I);
int __builtin_compact_to_i2x16_sat(long long int);
    SRs = COMPACT Rmd (IS);
int __builtin_compact_to_i2x16_trunc(long long int);
    SRs = COMPACT Rmd (T);
int __builtin_compact_to_i4x8(long long int);
    BRs = COMPACT SRmd (I);
long long int __builtin_expand_fr2x16(int);
    Rsd = EXPAND SRm;
long long int __builtin_expand_i2x16(int);
    Rsd = EXPAND SRm (I);
long long int __builtin_expand_i4x8(int);
    SRsd = EXPAND BRm (I);

```

Miscellaneous ALU Instructions:

```

long long int __builtin_llabs(long long int);
    LRsd = ABS Rmd;
long long int __builtin_llavg(long long int, long long int);
    LRsd = (Rmd + Rnd) / 2;

int __builtin_sum_2x16(int);

```

C/C++ Compiler Language Extensions

```
Rs = SUM Rm;
int __builtin_sum_2x32(long long int);
Rs = Rm + Rn;
int __builtin_sum_4x16(long long int);
Rs = SUM SRmd;
int __builtin_sum_4x8(int);
Rs = SUM Rm;
int __builtin_sum_8x8(long long int);
Rs = SUM SRmd;

long __builtin_lavg(long, long);
Rs = (Rm + Rn) / 2;
long __builtin_lavgt(long, long);
Rs = (Rm + Rn) / 2 (T);
long __builtin_lclip(long, long);
Rs = CLIP Rm BY Rn;
int __builtin_avg(int, int);
Rs = (Rm + Rn) / 2;
int __builtin_avgt(int, int);
Rs = (Rm + Rn) / 2 (T);

long long int __builtin_clip_4x16(long long int, long long int);
SRsd = CLIP Rmd by Rnd;
long long int __builtin_clip_8x8(long long int, long long int);
SRsd = CLIP Rmd by Rnd;
long long int __builtin_llclip(long long int, long long int);
LRsd = CLIP Rmd by Rnd;
int __builtin_clip(int, int);
Rs = CLIP Rm by Rn;
int __builtin_clip_2x16(int, int);
SRs = CLIP Rm by Rn;
int __builtin_clip_4x8(int, int);
SRs = CLIP Rm by Rn;

long long int __builtin_abs_4x16(long long int);
SRsd = ABS Rmd;
long long int __builtin_abs_8x8(long long int);
SRsd = ABS Rmd;
int __builtin_abs(int);
Rs = ABS Rm;
```

```

int __builtin_abs_2x16(int);
    SRs = ABS Rm;
int __builtin_abs_4x8(int);
    SRs = ABS Rm;
int __builtin_neg_2x16_sat(int);
    SRs = - Rm;
int __builtin_neg_sat(int);
    Rs = - Rm;

long long int __builtin_neg_4x16_sat(long long int);
    SRsd = - Rmd;
int __builtin_max(int, int);
    Rs = MAX (Rm, Rn);
int __builtin_min(int, int);
    Rs = MIN (Rm, Rn);
int __builtin_max_2x16(int, int);
    SRs = MAX (Rm, Rn);
int __builtin_min_2x16(int, int);
    SRs = MIN (Rm, Rn);
int __builtin_max_4x8(int, int);
    BRs = MAX (Rm, Rn);
int __builtin_min_4x8(int, int);
    BRs = MIN (Rm, Rn);

long long int __builtin_max_4x16(long long int, long long int);
    SRsd = MAX (Rmd, Rnd);
long long int __builtin_min_4x16(long long int, long long int);
    SRsd = MIN (Rmd, Rnd);
long long int __builtin_min_8x8(long long int, long long int);
    BRsd = MIN (Rmd, Rnd);
long long int __builtin_max_8x8(long long int, long long int);
    BRsd = MAX (Rmd, Rnd);
long long int __builtin_llmax(long long int, long long int);
    LRsd = MAX (Rmd, Rnd);
long long int __builtin_llmin(long long int, long long int);
    LRsd = MIN (Rmd, Rnd);

unsigned int __builtin_max_u2x16(unsigned int, unsigned int);
    SRs = MAX (Rm, Rn) (U);
unsigned int __builtin_min_u2x16(unsigned int, unsigned int);

```

C/C++ Compiler Language Extensions

```
SRs = MIN (Rm, Rn) (U);
unsigned long long int __builtin_max_u4x16(unsigned long long int,
                                           unsigned long long int);

SRsd = MAX (Rmd, Rnd);
unsigned long long int __builtin_min_u4x16(unsigned long long int,
                                           unsigned long long int);

SRsd = MIN (Rmd, Rnd) (U);
```

Shifter Instructions:

```
int __builtin_ashift_4x8(int, int);
BRs = ASHIFT Rm BY Rn;    or BRs = ASHIFT Rm BY imm4;
int __builtin_lshift_4x8(int, int);
BRs = LSHIFT Rm BY Rn;    or BRs = LSHIFT Rm BY imm4;
int __builtin_ashift_2x16(int, int);
SRs = ASHIFT Rm BY Rn;    or SRs = ASHIFT Rm BY imm5;
int __builtin_lshift_2x16(int, int);
SRs = LSHIFT Rm BY Rn;    or SRs = LSHIFT Rm BY imm5;
long long int __builtin_ashift_8x8(long long int, int);
BRsd = ASHIFT Rmd BY Rnd; or BRs = ASHIFT Rmd BY imm4;
long long int __builtin_lshift_8x8(long long int, int);
BRsd = LSHIFT Rmd BY Rnd; or BRs = LSHIFT Rmd BY imm4;
long long int __builtin_ashift_4x16(long long int, int);
SRsd = ASHIFT Rmd BY Rnd; or SRs = ASHIFT Rmd BY imm5;
long long int __builtin_lshift_4x16(long long int, int);
SRsd = LSHIFT Rmd BY Rnd; or SRs = LSHIFT Rmd BY imm5;
long long int __builtin_ashift_2x32(long long int, int);
Rsd = ASHIFT Rmd BY Rnd;  or Rs = ASHIFT Rmd BY imm6;
long long int __builtin_lshift_2x32(long long int, int);
Rsd = LSHIFT Rmd BY Rnd;  or Rs = LSHIFT Rmd BY imm6;
int __builtin_rotate_1x32(int, int);
Rs = ROT Rm BY Rn;        or Rs = ROT Rm BY imm6;
long long __builtin_rotate_2x32(long long, int);
Rsd = ROT Rmd BY Rnd;      or Rsd = ROT Rmd BY imm6;
long long __builtin_rotate_1x64(long long, int);
LRsd = ROT Rmd BY Rnd;     or LRsd = ROT Rmd BY imm6;
```

Bit Manipulation Instructions:

```

int __builtin_count_ones(int);
    Rs = ONES Rm;
int __builtin_lcount_ones(long);
    Rs = ONES Rm;
int __builtin_llcount_ones(long long int);
    Rs = ONES Rmd;
long long int __builtin_merge_2x16(int, int);
    SRsd = MERGE Rm, Rn;
long long int __builtin_merge_4x8(int, int);
    SRsd = MERGE Rm, Rn;
int __builtin_fdep(int, int, int);
    Rs += FDEP Rn BY Rm;
int __builtin_fdep_se(int, int, int);
    Rs += FDEP Rn BY Rm (SE);
int __builtin_fdep_zf(int, int, int);
    Rs += FDEP Rn BY Rm (ZF);
int __builtin_fdep_long_control(int, int, long long int);
    Rs += FDEP Rn BY Rmd;
int __builtin_fdep_long_control_se(int, int, long long int);
    Rs += FDEP Rn BY Rmd (SE);
int __builtin_fdep_long_control_zf(int, int, long long int);
    Rs += FDEP Rn BY Rmd (ZF);
long long int __builtin_fdep2(long long int, long long int, int);
    Rsd += FDEP Rnd BY Rm;
long long int __builtin_fdep2_se(long long int,
                                long long int, int);
    Rsd += FDEP Rnd BY Rm (SE);
long long int __builtin_fdep2_zf(long long int,
                                long long int, int);
    Rsd += FDEP Rnd BY Rm (ZF);
long long int __builtin_fdep2_long_control(long long int,
                                           long long int,
                                           long long int);
    Rsd += FDEP Rnd BY Rmd;
long long int __builtin_fdep2_long_control_se(long long int,
                                              long long int,
                                              long long int);
    Rsd += FDEP Rnd BY Rmd (SE);

```

C/C++ Compiler Language Extensions

```
long long int __builtin_fdep2_long_control_zf(long long int,
                                              long long int,
                                              long long int);

    Rsd += FDEP Rnd BY Rnd (ZF);
int __builtin_fext(int, int);
    Rs = FEXT Rm by Rn (SE);
int __builtin_fext_se(int, int);
    Rs = FEXT Rm by Rn (SE);
int __builtin_fext_ze(int, int);
    Rs = FEXT Rm by Rn;
int __builtin_fext_long_control(int, long long int);
    Rs = FEXT Rm by Rnd (SE);
int __builtin_fext_long_control_se(int, long long int);
    Rs = FEXT Rm by Rnd (SE);
int __builtin_fext_long_control_ze(int, long long int);
    Rs = FEXT Rm by Rnd;
long long int __builtin_fext2(long long int, int);
    Rsd = FEXT Rmd by Rn (SE);
long long int __builtin_fext2_se(long long int, int);
    Rsd = FEXT Rmd by Rn (SE);
long long int __builtin_fext2_ze(long long int, int);
    Rsd = FEXT Rmd by Rn;
long long int __builtin_fext2_long_control(long long int,
                                          long long int);

    Rsd = FEXT Rmd by Rnd (SE);
long long int __builtin_fext2_long_control_se(long long int,
                                          long long int);

    Rsd = FEXT Rmd by Rnd (SE);
long long int __builtin_fext2_long_control_ze(long long int,
                                          long long int);

    Rsd = FEXT Rmd by Rnd;
int __builtin_exp(int);
    Rs = EXP Rn;
int __builtin_exp2(long long int);
    Rs = EXP Rnd;
```

Multiplier Instructions:

```

unsigned int __builtin_mult_u2x16(unsigned int, unsigned int);
    Rsd = Rmd * Rnd (IU);
long long int __builtin_mult_i2x16_wide(int, int);
    Rsq = Rmd * Rnd (I);
long long int __builtin_mult_i2x16_wide_sat(int, int);
    Rsq = Rmd * Rnd (I);
unsigned long long int
    __builtin_mult_u2x16_wide(unsigned int,
                               unsigned int);

    Rsq = Rmd * Rnd (IU);
long long int __builtin_mult_i4x16(long long int, long long int);
    Rsd = Rmd * Rnd (I);
long long int __builtin_mult_i4x16_sat(long long int,
                                       long long int);

    Rsd = Rmd * Rnd (I);
unsigned long long int __builtin_mult_u4x16(unsigned long long int,
                                           unsigned long long int);

    Rsd = Rmd * Rnd (IU);
__builtin_quad __builtin_mult_i4x16_wide(long long int, long long int);
    Rsq = Rmd * Rnd (I);
__builtin_quad __builtin_mult_i4x16_wide_sat(long long int,
                                             long long int);

    Rsq = Rmd * Rnd (I);
__builtin_quad __builtin_mult_u4x16_wide(unsigned long long int,
                                           unsigned long long int);

    Rsq = Rmd * Rnd (IU);
int __builtin_frmulr(int, int);
    Rs = Rm * Rm;
int __builtin_frmulr_sat(int, int);
    Rs = Rm * Rm (S);
int __builtin_frmul(int, int);
    Rs = Rm * Rm (T);
int __builtin_frmul_sat(int, int);
    Rs = Rm * Rm (TS);
int __builtin_mult_i2x16(int, int);
    Rsd = Rmd * Rnd (I);
int __builtin_multr_fr2x16(int, int);
    Rsd = Rmd * Rnd;

```

C/C++ Compiler Language Extensions

```
int __builtin_multr_fr2x16_sat(int, int);
    Rsd = Rmd * Rnd (S);
int __builtin_mult_fr2x16(int, int);
    Rsd = Rmd * Rnd (T);
int __builtin_mult_fr2x16_sat(int, int);
    Rsd = Rmd * Rnd (TS);
long long int __builtin_mult_fr4x16(long long int, long long int);
    Rsd = Rmd * Rnd;
long long int __builtin_mult_fr4x16_sat(long long int,
                                       long long int);
    Rsd = Rmd * Rnd (S);
long long int __builtin_multr_fr4x16(long long int, long long int);
    Rsd = Rmd * Rnd;
long long int __builtin_multr_fr4x16_sat(long long int, long long int);
    Rsd = Rmd * Rnd (S);
int __builtin_cmult_i2x16(int, int);
    MRa += Rm ** Rn (I);
long long int __builtin_cmult_i2x16_wide(int, int);
    MRa += Rm ** Rn (I);
int __builtin_cmult_conj_i2x16(int, int);
    MRa += Rm ** Rn (IJ);
long long int __builtin_cmult_conj_i2x16_wide(int, int);
    MRa += Rm ** Rn (IJ);
int __builtin_cmult_fr2x16(int, int);
    MRa += Rm ** Rn;
int __builtin_cmult_conj_fr2x16(int, int);
    MRa += Rm ** Rn (J);
int __builtin_cmult_fr2x16_sat(int, int);
    MRa += Rm ** Rn;
```

Floating-Point Operations:

```
int __builtin_conv_FtoR(float);
    Rs = FIX FRm by 31;
float __builtin_conv_RtoF(int);
    FRs = FLOAT Rm by -31;
float __builtin_copysignf(float, float);
    FRs = Rm COPYSIGN Rn;
float __builtin_fabsf(float);
    FRs = ABS Rm;
```

```
float __builtin_favgf(float, float);
    FRs = (Rm + Rn) / 2;
float __builtin_fclipf(float, float);
    FRs = CLIP Rm by Rn;
float __builtin_fmaxf(float, float);
    FRs = MAX (Rm, Rn);
float __builtin_fminf(float, float);
    FRs = MIN (Rm, Rn);
```

Miscellaneous:

```
void __builtin_idle(void);
    IDLE;
void __builtin_idle_lp(void);
    IDLE (LP);
void __builtin_assert(int);    ignored
```

Memory Allocation:

```
void *__builtin_alloca_aligned(int size, int align);
```

Allocate “size” words of aligned data on the stack with alignment “align”.

Composition and Decomposition:

The following functions are used to compact or merge values to create larger or smaller types. In some cases, the optimizer is able remove instructions that may be generated by these functions.

To combine smaller objects into a single larger object, use:

```
__builtin_quad compose_128(long long ,long long);
    Compose a 128-bit datum
long long int __builtin_compose_64(int hi, int lo);
    Compose a 64-bit datum
unsigned long long compose_64u(unsigned int ,unsigned int);
    Compose an unsigned 64-bit datum
long double f_compose_64(float ,float);
    Compose a double precision datum from two single-precision
```

C/C++ Compiler Language Extensions

To create an object of size n by s ; n number of parts of size s bits, use

```
int compact_to_fr2x16(long long);  
    Create a 2x16 fractional item from a 2x32 fractional item  
int compact_to_i4x8(long long);  
    Create a 4x8 integer item from a 4x16 integer item
```

To expand an object of size n by s into the next size up, use:

Into two fracts:

```
long long expand_fr2x16(int );
```

Into two 2x16's:

```
long long expand_i4x8(int );
```

To extract the low/high half-out of various larger objects, use

```
int __builtin_low_32(long long int);  
    extract least significant word  
unsigned int __builtin_low_32u(unsigned long long int);  
    extract least significant word  
int __builtin_high_32(long long int);  
    extract most significant word  
unsigned int __builtin_high_32u(unsigned long long int);  
    extract most significant word  
long long int __builtin_low_64(__builtin_quad);  
    extract least significant words  
long long int __builtin_high_64(__builtin_quad);  
    extract most significant words  
float __builtin_f_low_32(long double);  
    extract least significant word from a 64-bit float  
float __builtin_f_high_32(long double);  
    extract most significant word from a 64-bit float
```

System Register Access:

To access to various system registers, use

```
int __builtin_sysreg_read(int);  
    read word-sized system register
```

```

long long int __builtin_sysreg_read2(int);
    read double-word-sized system register
__builtin_quad __builtin_sysreg_read4(int);
    read quad-word-sized system register
void __builtin_sysreg_write(int, unsigned int);
    write to word-sized system register
void __builtin_sysreg_write2(int, unsigned long long int);
    write to double-word-sized system register
void __builtin_sysreg_write4(int, __builtin_quad);
    write to quad-word-sized system register

```

 The register names are defined in `sysreg.h` and must be a literal used at compile time. The register numbers defined in `sysreg.h` are the compiler's internal register numbers. The effect of using the incorrect function for the size of the register or using a bad register number is undefined.

 Use of the `__XSTAT` and `__YSTAT` registers in the `sysreg` built-in functions are deprecated. There is no mechanism by which the source can associate a read of the `STAT` registers with a specific operation that may update the `STAT` registers. An inline asm can be used to get access to these registers if needed: the read of a `STAT` register can explicitly be placed where it is required.

Data Alignment Buffer (DAB) Built-in Functions

The DAB built-in functions provide access to the Data Alignment Buffer functionality. Two sets of built-in functions are provided. The first covers 32-bit data items and the built-ins are the same whether byte-addressing mode is enabled (with the “`-char-size-{8|32}`” switch) or not. The second set covers 16-bit data items and the details of the use depend on whether byte-addressing mode is enabled or not.

The following DAB built-in functions are provided.

```

long long __builtin_dab_2x32(int **a);
__builtin_quad __builtin_dab_4x32(int **a);
#ifdef __TS_BYTE_ADDRESS

```

C/C++ Compiler Language Extensions

```
// length given in terms of 16-bit items
int __builtin_dab_2x16(short **a);
long long __builtin_dab_4x16(short **a);
__builtin_quad __builtin_dab_8x16(short **a);
#else

// length given in terms of 16-bit items
int __builtin_dab_2x16(int **a, int offset);
long long __builtin_dab_4x16(int **a, int offset);
__builtin_quad __builtin_dab_8x16(int **a, int offset);
#endif
```

Every DAB built-in function takes a pointer to a pointer to the data as the first argument. The data pointer can in this way be automatically post-incremented by the execution of the built-in function according to the type and number of data items read. For the 16-bit DAB intrinsics, in byte-addressing mode the data pointer is a short integer pointer, and can thus address a given 16-bit quantity. In word-addressing mode, pointers do not have the resolution to address a part-word entity, so a second argument needs to be supplied which gives the half-word offset from the word-aligned address.

Here is an example of the 32-bit DAB built-in functions:

```
void func1 (__builtin_quad *a, int *b, int n) {
    int i;
    for (i=0; i<n; i++) {
        a[i] = __builtin_dab_4x32 (&b);
    }
}
```

Here is an example of the 16-bit DAB built-in functions in use in byte-addressing mode:

```
void func2 (long long int *a, short int *b, int n) {
    int i;
    for (i=0; i<n; i++) {
        a[i] = __builtin_dab_4x16 (&b);
    }
}
```

Here is an example of the 16-bit DAB built-in functions in use in word-addressing mode:

```
void func3 (long long int *a, int *b, int offset, int n) {
    int i;
    for (i=0; i<n; i++) {
        a[i] = __builtin_dab_4x16 (&b, offset);
    }
}
```

Circular Buffer Data Alignment Buffer (DAB) Built-in Functions

The circular buffer DAB intrinsics are similar to the DAB intrinsics (described in the [“Data Alignment Buffer \(DAB\) Built-in Functions” on page 1-129](#)), only with extra “*base*” and “*length*” parameters. The automatic post-increments now become post-increment with wrap-around according to the circular buffering.

Therefore,

```
long long __builtin_dabcb_2x32(int **a, int *base,
                               int length);
__builtin_quad __builtin_dabcb_4x32(int **a,
                                     int *base, int length);
#ifdef __TS_BYTE_ADDRESS
// length given in terms of 16-bit items
int __builtin_dabcb_2x16(short **a, short *base, int length);
long long __builtin_dabcb_4x16(short **a, short *base,
                                int length);
__builtin_quad __builtin_dabcb_8x16(short **a, short *base,
                                     int length);
#else
// length given in terms of 16-bit items
int __builtin_dabcb_2x16(int **a, int offset, int *base,
                        int length);
long long __builtin_dabcb_4x16(int **a, int offset,
                                int *base, int length);
```

C/C++ Compiler Language Extensions

```
__builtin_quad __builtin_dabcb_8x16(int **a, int offset,  
                                   int *base, int length);  
#endif
```

where “a” is the address of the pointer to be loaded from.

It should be noted that circular buffer with the DAB or SDAB only works when

- “base” is a quad-word aligned address, and
- the `length` is a multiple of quad-words.

The compiler may perform some checks on this where possible but in the case of non-constants cannot be expected to verify these conditions.

The compiler assumes that the initial value of `*a` or `*a + offset` is between `base` (inclusive) and `base+length` (not inclusive). In the case of the word-addressed compiler, circular-buffering ensures that the value of `*a + offset` is always in this range (and not necessarily `*a` itself).

Unoptimised, the code generated will perform two DAB accesses (the prime and the read) as well as initialisation and reset of the length and base registers for each DAB intrinsic. When optimisation is turned on, unnecessary setting of length and base registers and the DAB prime operation will be removed or hoisted out of loops to generate the optimal code.

Communications Logic Unit Operations

The following is a description of the built-in functions used to generate instructions to be executed by the Communications Logic Unit (CLU). They are grouped for clarity into `MAX / TMAX`, `PERMUTE`, `ACS`, `DESPREAD`, and `XCORRS`.

For instructions that have more than one output, there is a builtin for each of these outputs. In such cases the builtins for the second or third results need to be chained through the result of the first built-in function. For example, in the case of `DESPREAD`:

```
r = __builtin_despread (q, c, d);
th = __builtin_despread_res2 (r);
```

This is somewhat cumbersome and the compiler will be unforgiving if errors are introduced (such as forgetting the second built-in function), so the `builtins.h` file includes macros for each of the instructions that return multiple results. In the case of the example above, use the following form:

```
__despread (q, c, d, r, th);
```

The variables `r` and `th` are outputs even though they appear as arguments to `__despread`. Looking at the definition of `__despread` from `builtins.h` makes this more obvious:

```
#define __despread(I1,I2,I3,O1,O2) {      \
    O1 = __builtin_despread(I1,I2,I3);    \
    O2 = __builtin_despread_res2(O1);     \
}
```

In the examples for the instructions that have multiple results the short-hand form is used, and the inputs and outputs described.

TMAX, TMAX_ADD, TMAX_SUB, MAX_ADD, MAX_SUB

This section also covers the (S) variants of these instructions which deal with 16-bit components.

Instruction:

```
Rs = TMAX (TRm, TRn)
```

Builtin:

```
int __builtin_tmax (int, int);
```

Example:

```
int r, a, b;
r = __builtin_tmax (a, b);
```

Description:

r corresponds to Rs
a corresponds to TRm
b corresponds to TRn

Instruction:

SRs = TMAX (TRm, TRn)

Builtin:

```
int __builtin_tmax_4s (int, int);
```

Example:

```
int r, a, b;  
r = __builtin_tmax_4s (a, b);
```

Description:

r corresponds to Rs
a corresponds to TRm
b corresponds to TRn

Instruction:

TRsd = TMAX (TRmd + Rmq_h, TRnd + Rmq_l)

Builtin:

```
long long int __builtin_tmax_add (long long int,  
                                  long long int,  
                                  __builtin_quad);
```

Example:

```
long long int r, a, b;  
__builtin_quad q;  
r = __builtin_tmax_add (a, b, q);
```

Description:

r corresponds to TRsd
a corresponds to TRmd

b corresponds to TRnd

q corresponds to Rmq

Instruction:

TRsd = TMAX (TRmd - Rmq_h, TRnd - Rmq_l)

Builtin:

```
long long int __builtin_tmax_sub (long long int,
                                long long int,
                                __builtin_quad);
```

Example:

```
long long int r, a, b;
__builtin_quad q;
r = __builtin_tmax_sub (a, b, q);
```

Description:

r corresponds to TRsd

a corresponds to TRmd

b corresponds to TRnd

q corresponds to Rmq

Instruction:

TRsd = MAX (TRmd + Rmq_h, TRnd + Rmq_l)

Builtin:

```
long long int __builtin_max_add (long long int,
                                long long int,
                                __builtin_quad);
```

Example:

```
long long int r, a, b;
__builtin_quad q;
r = __builtin_max_add (a, b, q);
```

Description:

r corresponds to TRsd

C/C++ Compiler Language Extensions

a corresponds to TRmd

b corresponds to TRnd

q corresponds to Rmq

Instruction:

TRsd = MAX (TRmd - Rmq_h, TRnd - Rmq_l)

Builtin:

```
long long int __builtin_max_sub (long long int,  
                                long long int,  
                                __builtin_quad);
```

Example:

```
long long int r, a, b;  
__builtin_quad q;  
r = __builtin_max_sub (a, b, q);
```

Description:

r corresponds to TRsd

a corresponds to TRmd

b corresponds to TRnd

q corresponds to Rmq

Instruction:

STRsd = TMAX (TRmd + Rmq_h, TRnd + Rmq_l)

Builtin:

```
long long int __builtin_tmax_add_8s (long long int,  
                                     long long int,  
                                     __builtin_quad);
```

Example:

```
long long int r, a, b;  
__builtin_quad q;  
r = __builtin_tmax_add_8s (a, b, q);
```

Description:

r corresponds to TRsd

a corresponds to TRmd

b corresponds to TRnd

q corresponds to Rmq

Instruction:

$$STRsd = TMAX (TRmd - Rmq_h, TRnd - Rmq_l)$$
Builtin:

```
long long int __builtin_tmax_sub_8s (long long int,
                                     long long int,
                                     __builtin_quad);
```

Example:

```
long long int r, a, b;
__builtin_quad q;
r = __builtin_tmax_sub_8s (a, b, q);
```

Description:

r corresponds to TRsd

a corresponds to TRmd

b corresponds to TRnd

q corresponds to Rmq

Instruction:

$$STRsd = MAX (TRmd + Rmq_h, TRnd + Rmq_l)$$
Builtin:

```
long long int __builtin_max_add_8s (long long int,
                                     long long int,
                                     __builtin_quad);
```

Example:

```
long long int r, a, b;
```

C/C++ Compiler Language Extensions

```
__builtin_quad q;  
r = __builtin_max_add_8s (a, b, q);
```

Description:

r corresponds to TRsd
a corresponds to TRmd
b corresponds to TRnd
q corresponds to Rmq

Instruction:

STRsd = MAX (TRmd - Rmq_h, TRnd - Rmq_l)

Builtin:

```
long long int __builtin_max_sub_8s (long long int,  
                                     long long int,  
                                     __builtin_quad);
```

Example:

```
long long int r, a, b;  
__builtin_quad q;  
r = __builtin_max_sub_8s (a, b, q);
```

Description:

r corresponds to TRsd
a corresponds to TRmd
b corresponds to TRnd
q corresponds to Rmq

PERMUTE

Instruction:

Rsd = PERMUTE (Rmd, Rn)

Builtin:

```
long long int __builtin_permute_8b (long long int, int);
```

Example:

```
long long int r, a;
int b;
r = __builtin_permute_8b (a, b);
```

Description:

r corresponds to Rsd

a corresponds to Rmd

b corresponds to Rnd

Instruction:

Rsq = PERMUTE (Rmd, -Rmd, Rn)

Builtin:

```
__builtin_quad __builtin_permute_8s (long long int, int);
```

Example:

```
__builtin_quad r;
long long int a;
int b;
r = __builtin_permute_8s (a, b);
```

Description:

r corresponds to Rsd

a corresponds to Rmd

b corresponds to Rnd

ACS

Instruction:

TRsq = ACS (TRmd, TRnd, Rm) (TMAX)

Builtins:

```
__builtin_quad __builtin_acs_tmax (long long int,
                                   long long int,
                                   int,
```

C/C++ Compiler Language Extensions

```
                                long long int);  
long long int __builtin_acs_res2 (__builtin_quad);
```

Shorthand:

```
__acs_tmax (long long int,  
            long long int,  
            int,  
            long long int,  
            __builtin_quad,  
            long long int);
```

Example:

```
__builtin_quad r;  
long long int a, b, thi, tho;  
int c;  
__acs_tmax (a, b, c, thi, r, tho);
```

Description:

a corresponds to TRmd

b corresponds to TRnd

c corresponds to Rm

thi corresponds to Trellis History Registers before execution (input)

r corresponds to TRsq

tho corresponds to Trellis History Registers after execution (output)

Instruction:

TRsq = ACS (TRmd, TRnd, Rm)

Builtins:

```
__builtin_quad __builtin_acs_max (long long int,  
                                   long long int,  
                                   int,  
                                   long long int);  
long long int __builtin_acs_res2 (__builtin_quad);
```

Shorthand:

```
__acs_max (long long int,
          long long int,
          int,
          long long int,
          __builtin_quad,
          long long int);
```

Example:

```
__builtin_quad r;
long long int a, b, thi, tho;
int c;
__acs_max (a, b, c, thi, r, tho);
```

Description:

a corresponds to TRmd

b corresponds to TRnd

c corresponds to Rm

thi corresponds to Trellis History Registers before execution (input)

r corresponds to TRsq

tho corresponds to Trellis History Registers after execution (output)

Instruction:

STRsq = ACS (TRmd, TRnd, Rm) (TMAX)

Builtins:

```
__builtin_quad __builtin_acs_tmax_8s (long long int,
                                     long long int,
                                     int,
                                     long long int);

long long int __builtin_acs_res2 (__builtin_quad);
```

Shorthand:

```
__acs_tmax_8s (long long int,
              long long int,
              int,
```

C/C++ Compiler Language Extensions

```
long long int,  
__builtin_quad,  
long long int);
```

Example:

```
__builtin_quad r;  
long long int a, b, thi, tho;  
int c;  
__acs_tmax_8s (a, b, c, thi, r, tho);
```

Description:

a corresponds to TRmd
b corresponds to TRnd
c corresponds to Rm
thi corresponds to Trellis History Registers before execution (input)
r corresponds to TRsq
tho corresponds to Trellis History Registers after execution (output)

Instruction:

STRsq = ACS (TRmd, TRnd, Rm)

Builtins:

```
__builtin_quad __builtin_acs_max_8s (long long int,  
                                     long long int,  
                                     int,  
                                     long long int);  
long long int __builtin_acs_res2 (__builtin_quad);
```

Shorthand:

```
__acs_max_8s (long long int,  
             long long int,  
             int,  
             long long int,  
             __builtin_quad,  
             long long int);
```

Example:

```
__builtin_quad r;
long long int a, b, thi, tho;
int c;
__acs_max_8s (a, b, c, thi, r, tho);
```

Description:

a corresponds to TRmd

b corresponds to TRnd

c corresponds to Rm

thi corresponds to Trellis History Registers before execution (input)

r corresponds to TRsq

tho corresponds to Trellis History Registers after execution (output)

DESPREAD**Instruction (ADSP-TS101 processor):**

$$TRs = \text{DESPREAD} (Rmq, THrd) + TRn$$
Instruction (ADSP-TS201 processor):

$$TRs += \text{DESPREAD} (Rmq, THrd)$$
Builtins:

```
int __builtin_despread (__builtin_quad, long long int, int);
long long int __builtin_despread_res2 (int);
```

Shorthand:

```
__despread (__builtin_quad,
            long long int,
            int,int,
            long long int);
```

Example:

```
__builtin_quad q;
long long int thi, tho;
int d;
```

C/C++ Compiler Language Extensions

```
int r;  
__despread (q, thi, d, r, tho);
```

Description:

q corresponds to Rmq
thi corresponds to Trellis History Registers before execution (input)
d corresponds to TRn
r corresponds to TRs
tho corresponds to Trellis History Registers after execution (output)

Instruction (ADSP-TS101 processor):

TRs = DESPREAD (Rmq, THrd) + TRn

Instruction (ADSP-TS201 processor):

TRs += DESPREAD (Rmq, THrd)

Builtins:

```
int __builtin_despread_i (__builtin_quad, long long int, int);  
long long int __builtin_despread_res2 (int);
```

Shorthand:

```
__despread_i (__builtin_quad,  
              long long int,  
              int,int,  
              long long int);
```

Example:

```
__builtin_quad q;  
long long int thi, tho;  
int d;  
int r;  
__despread (q, thi, d, r, tho);
```

Description:

In this version, THrd is loaded using the (i) option to interleave the input bits. Obviously, when loading the spreading codes interleaved into the Trellis History registers for the first DESPREAD, the __despread_i built-in

function should be used. When chaining the shifted spreading codes from the first DESPREAD into a subsequent DESPREAD, use the `__despread` built-in function, otherwise the Trellis History registers after the first DESPREAD will be stored and then reloaded (with interleave) again.

`q` corresponds to `Rmq`

`thi` corresponds to Trellis History Registers before execution (input)

`d` corresponds to `TRn`

`r` corresponds to `TRs`

`tho` corresponds to Trellis History Registers after execution (output)

XCORRS

Instruction (ADSP-TS201 processor):

`TR15:0/31:16 = XCORRS (Rmq, THRq) (CUT #imm) (CLR) (EXT)`

Builtins:

```
__builtin_quad __builtin_xcorrs (__builtin_quad signal,
                                long long int codeslo,
                                long long int codeshi,
                                int cut,
                                __builtin_quad accum0,
                                __builtin_quad accum1,
                                __builtin_quad accum2,
                                __builtin_quad accum3);
__builtin_quad __builtin_xcorrs_res2 (__builtin_quad);
__builtin_quad __builtin_xcorrs_res3 (__builtin_quad);
__builtin_quad __builtin_xcorrs_res4 (__builtin_quad);
long long int __builtin_xcorrs_res5 (__builtin_quad);
long long int __builtin_xcorrs_res6 (__builtin_quad);
```

Shorthand:

```
void __xcorrs (__builtin_quad signal,
              long long int *codeslo,
              long long int *codeshi,
              int cut,
              int *accum);
```

Example:

```
__builtin_quad signal;  
long long int codeslo;  
long long int codeshi;  
__builtin_quad accums[4];  
__xcorrs (signal, &codeslo, &codeshi, /* cut */ 0, &accums);
```

Description:

`signal` corresponds to `Rmq`

`codeslo` and `codeshi` correspond to low and high halves of `THRq` (input and output)

`accums` corresponds to `TR15:0` or `TR31:16` depending on which half of the register file is being used.

The contents of `accums` will be copied into the Trellis registers before the instruction, and the accumulations copied back into `accums` after execution. The compiler will optimise sequences of `XCORRS` to remove unnecessary transfers in and out of the Trellis registers, as it does for the other enhanced communication built-in functions.

The same thing will happen for the Trellis History registers, which are loaded from the two double-words pointed to by `codeslo` and `codeshi`, and stored back into these locations after execution.

The `accums` pointer must be quad aligned, and the `codeslo` and `codeshi` pointers must be double-word aligned. These alignments should be declared with `__builtin_aligned` where necessary. The behaviour of the compiler, and the resulting code, will be undefined if this is not done.

The `cut` argument must be an immediate value.

The following shorthand forms will be provided for variants of the `XCORRS` instruction which specify the (EXT) and/or (CLR) options.

```

void __xcorrs_clr (__builtin_quad signal,
                  long long int *codeslo,
                  long long int *codeshi,
                  int cut,
                  int *accum);
void __xcorrs_ext (__builtin_quad signal,
                  long long int *codeslo,
                  long long int *codeshi,
                  int cut,
                  int *accum);
void __xcorrs_clr_ext (__builtin_quad signal,
                      long long int *codeslo,
                      long long int *codeshi,
                      int cut,
                      int *accum);

```

At a low level, these macros make use of alternative built-in functions for the primary result and the same built-in functions as above for the subsequent results.

There are also `_i` versions of the built-in function:

```

void __xcorrs_i (__builtin_quad signal,
                 long long int *codeslo,
                 long long int *codeshi,
                 int cut,
                 int *accum);
void __xcorrs_i_clr (__builtin_quad signal,
                    long long int *codeslo,
                    long long int *codeshi,
                    int cut,
                    int *accum);
void __xcorrs_i_ext (__builtin_quad signal,
                    long long int *codeslo,
                    long long int *codeshi,
                    int cut,
                    int *accum);
void __xcorrs_i_clr_ext (__builtin_quad signal,
                        long long int *codeslo,
                        long long int *codeshi,

```

```
int cut,  
int *accum);
```

For the `_i` versions, the `codeshi` value is loaded into the upper Trellis History registers with the `(i)` option, to interleave the input bits of the spreading code.

Pragmas

The compiler supports a number of pragmas. Pragmas are implementation-specific directives that modify the compiler's behavior. There are two types of pragma usage: pragma directives and pragma operators.

Pragma directives have the following syntax:

```
#pragma pragma-directive pragma-directive-operands new-line
```

Pragma operators have the following syntax:

```
_Pragma ( string-literal )
```

When processing a pragma operator, the compiler effectively turns it into a pragma directive using a non-string version of *string-literal*. This means that the following pragma directive

```
#pragma linkage_name mylinkname
```

can also be equivalently be expressed using the following pragma operator

```
_Pragma ( "linkage_name mylinkname" )
```

The examples in this manual use the directive form.

The C/C++ compiler issues a warning when it encounters an unrecognized pragma directive or pragma operator. The C/C++ compiler does not expand any pre-processor macros used within any pragma directive or pragma operator.

The C compiler supports pragmas for:

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

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

- [“Data Alignment Pragmas” on page 1-150](#)
- [“Interrupt Handler Pragmas” on page 1-151](#)
- [“Loop Optimization Pragmas” on page 1-154](#)
- [“Function Side-Effect Pragmas” on page 1-157](#)
- [“General Optimization Pragmas” on page 1-162](#)
- [“Linking Control Pragmas” on page 1-163](#)
- [“Template Instantiation Pragmas” on page 1-165](#)

Refer to Chapter 2, [“Achieving Optimal Performance from C/C++ Source Code”](#), on how to use pragmas for code optimization.

Data Alignment Pragmas

The data alignment pragma is used to modify how the compiler arranges data within memory. Since the TigerSHARC processor architecture requires memory accesses to be naturally aligned, each data item is normally aligned at least as strongly as itself—double-word items have an alignment of 2 and quad-word items have an alignment of 4.

In byte-addressing mode, the same rule applies—two-byte `shorts` have an alignment of 2 (bytes) and four-byte `longs` have an alignment of 4 (bytes). When structs are defined, the struct's overall alignment is the same as the field which has the largest alignment. The struct's size may need padding to ensure all fields are properly aligned, and that the struct's overall size is a multiple of its alignment.

Sometimes, it is useful to change these alignments, for instance a struct may have its alignment increased to improve the compiler's opportunities in vectorizing access to the data.

Alignments specified for data alignment must be a power of 2.



Note that the minimum alignment allowed for structs in byte-addressing mode is 4 (32-bits).

`#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. The pragma's effect is that the next variable or field declaration will be aligned on a boundary specified by *num*.

- If *num* is greater than the alignment normally required by the following variable or field declaration, then the variable or field declaration's alignment is changed to be *num*.

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

```
typedef struct {
    #pragma align 4
    int foo;
    int bar;
    #pragma align 4
    int baz;
} aligned_ints;
```

In this example, any object declared of type `aligned_ints` will have the `foo` and `baz` members aligned on quad-word boundaries. The size of this structure will be eight words with two words unused between `bar` and `baz`, and three words of padding at the end of the structure. In byte addressing mode, the `NUM` argument to `align` would need to be 16 and not 4, as the argument to `align` is in addressable units.

Interrupt Handler Pragas

The interrupt pragmas provide a method by which the user can write interrupt service routines in C and install them directly into the interrupt vector table, bypassing the dispatcher provided with the C run time library.

Marking a routine with an `interrupt` pragma causes the compiler to save all necessary state at the beginning of the routine and restore it at the end of the routine. It also causes the compiler to emit the corresponding code to return from the service routine correctly.

There are two interrupt pragmas provided by the compiler for the TigerSHARC processors. The `interrupt` pragma is used to define a non-reentrant interrupt handler (one which cannot be interrupted by a subsequent interrupt). The `interrupt_reentrant` pragma is used to define a reentrant interrupt handler (one which can be interrupted by higher-priority interrupts).

To define an interrupt handler in C whose address can be put directly into the interrupt vector table, put the relevant pragma prior to the function definition (see the example in this section).

-  It is possible to use the interrupt dispatcher in the default C run time library to register an interrupt or an exception handler and in the same application install a routine marked with the 'interrupt' pragma into other entries in the vector table.
-  It is not possible to raise an interrupt defined in this way using the `raise` function as this is currently only provided to raise interrupts registered with the interrupt dispatcher.

Interrupt Pragma Example:

This program is designed to run on an ADSP-TS101 EZ-KIT Lite board. It flashes the **FLAG2** LED (for the given processor on which it is running), and pressing the **IRQ0** button (for the same processor) causes the rate of flash to change. These two actions are handled by interrupts, the service routines for which are written in C using the `interrupt` pragma. While this is happening, the main loop of the program produces some output.

```
#include <stdio.h>

#include <builtins.h>
#include <sysreg.h>
#include <defts101.h>

#define SQCTL_TMRORN MAKE_BITMASK_(SQCTL_TMRORN_P)

#pragma interrupt
void timer0h_isr (void) {
    static int led = 0;
    led = !led;
    if (led == 1) {
        __builtin_sysreg_write(__SQCTLST, SQCTL_FLAG2_OUT);
    } else {
        __builtin_sysreg_write(__SQCTLCL, ~SQCTL_FLAG2_OUT);
    }
}
```

```

}

#pragma interrupt
void irq0_isr (void) {
    static int speed = 0;

    speed = !speed;

    if (speed == 1) {
        __builtin_sysreg_write (__TMRINOH, 0);
        __builtin_sysreg_write (__TMRINOL, 250000000);
    } else {
        __builtin_sysreg_write (__TMRINOH, 0);
        __builtin_sysreg_write (__TMRINOL, 125000000);
    }
}

int main (void) {
    int r;

    /* Enable output on FLAG2 (to allow the LED state to
       be set on EZ-KIT hardware */
    __builtin_sysreg_write (__SQCTLST, SQCTL_FLAG2_EN);

    /* Install the two service routines into the vector table */
    __builtin_sysreg_write (__IVTIMER0HP, (int) timer0h_isr);
    __builtin_sysreg_write (__IVIRQ0, (int) irq0_isr);

    /* Set Timer 0 to count 125 million cycles */
    __builtin_sysreg_write (__TMRINOH, 0);
    __builtin_sysreg_write (__TMRINOL, 125000000);

    /* Make the IRQ0 interrupt edge-sensitive */
    __builtin_sysreg_write (__SQCTLCL, ~(1<<SQCTL_IRQ0_EDGE_P));

    /* Set Timer 0 running */
    __builtin_sysreg_write (__SQCTLST, (1 << SQCTL_TMRORN_P));

    /* Set the global interrupt enable bit and the IRQ0 and
       TIMER0HP bits in the interrupt mask register */
    r = __builtin_sysreg_read (__IMASKH);
    r |= (1<<INT_GIE_P) | (1<<INT_IRQ0_P) | (1<<INT_TIMER0H_P);
    __builtin_sysreg_write (__IMASKH, r);

```

```
/* While waiting for interrupts, produce some output */
for (r=0; r<0x7fffffff; r++) {
    printf ("Loop %d\n", r);
}
```

Loop Optimization Pragas

Loop optimization pragmas give the compiler additional information about usage within a particular loop, which allows the compiler to perform more aggressive optimization. The pragmas are placed before the loop statement, and apply to the statement that immediately follows, which must be a `for`, `while` or `do` statement to have effect. In general, it is most effective to apply loop pragmas to inner-most loops, since the compiler can achieve the most savings there.

#pragma all_aligned

The `#pragma all_aligned` tells the compiler that all pointer induction variables in the loop are initially aligned to the maximum interesting alignment of the architecture. For TigerSHARC processors, that is quad-word aligned. The pragma takes an optional argument (*n*) which can specify that the pointers are aligned after *n* iterations. Therefore, the `#pragma all_aligned(1)` says that after one iteration, all the pointer induction variables of the loop are so aligned. In other words, the default argument is zero.

#pragma no_vectorization

The `#pragma no_vectorization` pragma is used to turn off all vectorization for the loop on which it is specified.

#pragma loop_count(*min*, *max*, *modulo*)

The `#pragma loop_count(min, max, modulo)` appears just before the loop it describes. It asserts that the loop will iterate at least *min* times, no more than *max* times, and a multiple of *modulo* times. This information enables

the optimizer to omit loop guards and to decide whether the loop is worth completely unrolling and whether code needs to be generated for odd iterations. The last two arguments can be omitted if they are unknown.

For example,

```
int i;
#pragma loop_count(24, 48, 8)
for (i=0; i < n; i++)
```

The `#pragma must_iterate(min, max, modulo)` is also accepted for compatibility with other compilers.

#pragma different_banks

The `#pragma different_banks` allows the compiler to assume that groups of memory accesses based on different pointers within a loop reside in different memory banks. By scheduling them together, memory access is much improved.

#pragma vector_for

The `#pragma vector_for` notifies the compiler that it is safe to execute some number of consecutive iterations of the loop in parallel. Therefore, the form `#pragma vector_for (number)` notifies the compiler that a *number* of consecutive iterations maybe executed in parallel. The form `#pragma vector_for (without the number)` tells the compiler that any number of consecutive iterations maybe executed in parallel.

The `vector_for` pragma does not force the compiler to vectorize the loop; the optimizer checks various properties of the loop and does not vectorize it if it believes it is unsafe or if it cannot deduce that the various properties necessary for the vectorization transformation are valid.

Strictly speaking, the pragma simply disables checking for loop-carried dependencies.

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

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



The `#pragma SIMD_for` may be used as an alternative to `#pragma vector_for(2)`.

#pragma no_alias

Use `#pragma no_alias` to tell the compiler the following loop has no loads or stores that conflict due to references to the same location through different pointers, known as “aliases”.

In the example,

```
void vadd(int *a, int *b, int *out, int n) {  
    int i;  
    #pragma no_alias  
        for (i=0; i < n; i++)  
            out[i] = a[i] + b[i];  
}
```

the use of the `no_alias` pragma just before the loop informs the compiler that the pointers *a*, *b* and *out* point to different arrays, so no load from *b* or *a* will be using the same address as any store to *out*. Therefore, `a[i]` or `b[i]` is never an alias for `out[i]`.

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

Function Side-Effect Pragmas

The function side-effect pragmas (`pure`, `const`, and `regs_clobbered`) are used before a function declaration to give the compiler additional information about the function in order to enable it to improve the code surrounding the function call. These pragmas should be placed before a function declaration and should apply to that function. For example,

```
#pragma pure
long dot(short*, short*, int);
```

#pragma pure

The `#pragma pure` tells the compiler that the function does not write to any global variables, and does not read or write any volatile variables. Its result, therefore, is a function of its parameters or of global variables. If any of the parameters are pointers, the function may read the data they point at but it may not write it.

As this means the function call will have the same effect every time it is called, between assignments to global variables, the compiler need not generate the code for every call. Therefore, in this example,

```
#pragma pure
long sdot(short *, short *, int);

long tendots(short *a, short *b, int n) {
    int i;
    long s = 0;
    for (i = 1; i < 10; ++i)
        s += sdot(a, b, n); // call can get hoisted out of loop
    return s;}

```

the compiler can replace the ten calls to `sdot` with a single call made before the loop.

#pragma const

The `#pragma const` is a more restrictive form of the `pure pragma`. It tells the compiler that the function does not read from global variables as well as not writing to them or reading or writing volatile variables. The result of the function is therefore a function of its parameters. If any of the parameters are pointers, the function may not even read the data they point at.

#pragma regs_clobbered *string*

The `#pragma regs_clobbered string` may be used with a function declaration or definition to specify which registers are modified (or clobbered) by that function. The *string* contains a list of registers and is case-insensitive.

When used with an external function declaration, this pragma acts as an assertion telling the compiler something it would not be able to discover for itself. In the example,

```
#pragma regs_clobbered "xr5 xr8 j6"
void f(void);
```

the compiler will know that only registers `r5`, `r8` and `i6` may be modified by the call to `f`, so it may keep local variables in other registers across that call.

The `regs_clobbered` pragma may also be used with a function definition, or a declaration preceding a definition, when it acts as a command to the compiler to generate register saves and restores on entry and exit from the function to ensure it only modifies the registers in *string*. For example,

```
#pragma regs_clobbered "j0 j4"
int g(int a) {
```

```

    return a+3;
}

```

String Syntax

A `regs_clobbered` string consists of a list of registers, register ranges, or register sets that are clobbered (Table 1-12). The list is separated by spaces, commas, or semicolons.

A *register* is a single register name, which is the same as that which may be used in an assembly file. Register names (when not used in ranges) may be pairs, quads or SIMD. Therefore, “r1”, “xvr2”, “vr3:0” and “r3:0” are acceptable register names specifying register sets “xr1,vr1”, “xr2,vr2”, “vr0,vr1,vr2,vr3” and “xr0,xr1,xr2,xr3,vr0,vr1,vr2,vr3”, respectively.

A *register range* consists of start and end registers which both reside in the same register class, separated by a hyphen. All registers between the two (inclusive) are clobbered.

A *register set* is a name for a specific set of commonly clobbered registers that is predefined by the compiler. Table 1-12 shows defined register sets,

Table 1-12. Clobbered Register Sets

Set	Registers
CCset	XSTAT, YSTAT, JSTAT, KSTAT
MRset	XMR0-XMR4, YMR0-YMR4
PRset	XPR0, XPR1, YPR0, YPR1
LCset	LC0, LC1
ACCELset	XTR0-XTR31, YTR0-YTR31, XTHR0-XTHR3, YTHR0-YTHR3
DABset	XDAB0-XDAB3, YDAB0-YDAB3
CBset	J0-J3, K0-K3, JL0-JL3, KL0-KL3, JB0-JB3, KB0-KB3
Xscratch	Members of X-registers that are scratch by default, XSTAT

Table 1-12. Clobbered Register Sets (Cont'd)

Set	Registers
Yscratch	Members of Y-registers that are scratch by default, YSTAT
XYscratch	Xscratch , Yscratch
Jscratch	Members of J-registers that are scratch by default, JSTAT
Kscratch	Members of K-registers that are scratch by default, KSTAT
JKscratch	Jscratch , Kscratch
XYJKscratch	JKscratch , XYscratch
ALLscratch	Entire default volatile set

The strings are also valid as GNU `asm` clobbered sets.

When the compiler detects an illegal string, a warning is issued and the default volatile set as defined in this compiler manual is used instead.

Unclobberable and Must Clobber Registers

There are certain caveats as to what registers may or must be placed in the clobbered set. On TigerSHARC processors, the registers J26, J27, K26 and K27 may not be specified in the clobbered set, as the correct operation of the function call requires their value to be preserved. If the user specifies them in the clobbered set, a warning will be issued and they will be removed from the specified clobbered set.

Registers from these classes,

X, Y, J, K, XT, YT, XTH, YTH, XDAB, YDAB, LC, JB, KB, JL, KL,
XMR, YMR, XP, YP, XSTAT, YSTAT, JSTAT, KSTAT

may be specified in the clobbered set and code will be generated to save them as necessary. All other registers are never preserved whether specified as clobbered or not. They are not automatically used by the compiler. It is assumed that the user, should they change the register, will want the change preserved.

Function Return Registers

Function return registers are visible to the caller. They may or may not appear in the clobbered set of the callee (it will make no difference to the generated code; the return register will never be saved and restored). Only the return register used by the particular function return type is special. Return registers used by different return types will be treated in the clobbered list in the conventional way. For example,

```
int f();
/* returns via J8. XR8 and XR9 may be preserved across call */
long long f();
/* returns via XR9:8. J8 may be preserved across call      */
void f();
/* J8, XR8 and XR9 may all be preserved across the call    */
```

Function Parameters

Function calling conventions are visible to the caller and do not affect the clobbered set that may be used on a function. For example,

```
#pragma regs_clobbered ""    // clobbers nothing
void f(int a, int b);
void g() {
    f(2,3);
}
```

The parameters *a* and *b* are passed in registers registers J4 and J5 respectively. No matter what happens in function *f*, the values of J4 and J5 after the call will still be 2 and 3, respectively.

Pragma Interrupt Restriction

The `regs_clobbered` pragma may not be used in conjunction with `#pragma interrupt`. If both are specified, a warning is issued and the `regs_clobbered` pragma is ignored.

Best Usage

To obtain best results with the pragma, it is best to restrict the clobbered set to be a subset of the default volatile set. The compiler is likely to produce more efficient code this way than if the volatile set is changed to use

the same number of registers but which do not make a subset of the default volatile set. For example, if the default volatile set is “r0-r2, r5-r7”, then using a `regs_clobbered` set of “r0-r2” will likely give better results than using “r3-r5”.

While considering when to apply the `regs_clobbered` pragma, it may be useful to look at the output of the compiler to see how many scratch registers were used. Restricting the volatile set to these registers will produce no impact on the code produced for the function but may free up registers for the caller to allocate across the call site.

General Optimization Pragas

The compiler supports several pragmas which can change the optimization level while a given module is being compiled. These pragmas must be used globally, immediately prior to a function definition. The pragmas do not just apply to the immediately following function; they remain in effect until the end of the compilation, or until superceded by one of the following `optimize_` pragmas.

- **`#pragma optimize_off`**

This pragma turns off the optimizer, if it was enabled. High-level optimizations, such as inlining and constant-expression evaluation, will still apply. This pragma has no effect if Interprocedural Optimization Analysis is enabled.

- **`#pragma optimize_for_space`**

This pragma turns the optimizer back on, if it was disabled, or sets the focus to give reduced code size a higher priority than high performance, where these conflict.

- **`#pragma optimize_for_speed`**

This pragma turns the optimizer back on, if it was disabled, or sets the focus to give high performance a higher priority than reduced code size, where these conflict.

- **#pragma optimize_as_cmd_line**

This pragma resets the optimization settings to be those specified on the `ccts` command line when the compiler was invoked.

These are code examples for the `optimize_` pragmas.

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

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

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

Linking Control Pragmas

Linking pragmas (`linkage_name`, `retain_name` and `#pragma weak_entry`) change how a given global function or variable is viewed during the linking stage.

#pragma linkage_name *identifier*

The `#pragma linkage_name` associates the *identifier* with the next external function declaration. It ensures that the *identifier* is used as the external reference, instead of following the compiler's usual conventions. If the *identifier* is not a valid function name, as could be used in normal function definitions, the compiler will generate an error. See also the `asm` keyword (described [on page 1-178](#)).

The following shows an example use of this pragma.

```
#pragma linkage_name realfuncname
void funcname ();
void func() {
    funcname(); /* compiler will generate a call to realfuncname */
}
```

#pragma retain_name

The `#pragma retain_name` indicates that the external function or variable declaration that follows the pragma is not removed even though Interprocedural Analysis (IPA) sees that it is not used. For example, use this pragma for C functions that are called only from an assembler routine, such as the start-up code sequence invoked before `main()`.

The following example shows how to use this pragma.

```
int delete_me(int x) {
    return x-2;
}

#pragma retain_name
int keep_me(int y) {
    return y+2;
}

int main(void) {
    return 0;
}
```

Since the program has no uses of either `delete_me()` or `keep_me()`, the compiler will remove `delete_me()`, but will keep `keep_me()` because of the pragma. You do not need to specify `retain_name` for `main()`.

For more information on IPA, see [“Interprocedural Analysis” on page 1-62](#).

#pragma weak_entry

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

The following are example uses of the `pragma weak_entry` directive.

```
#pragma weak_entry
int w_var = 0;

#pragma weak_entry
void w_func(){}

```

Template Instantiation Pragas

The template instantiation pragmas (`__attribute__((__weak_entry__))`, `__attribute__((__do_not_instantiate__))` and `__attribute__((__can_instantiate__))`) give fine grain control over where (that is, in which object file) the individual instances of template functions, and member functions and static members of template classes are created. The creation of these instances from a template is known in “C++ speak” as instantiation. As templates are a feature of C++, these pragmas are allowed only in `-c++` mode.

These pragmas take the name of an instance as a parameter, as shown in [Table 1-13](#).

Table 1-13. Instance Names

Name	Parameter
a template class name	<code>A<int></code>
a template class declaration	<code>class A<int></code>
a member function name	<code>A<int>::f</code>
a static data member name	<code>A<int>::I</code>
a static data declaration	<code>int A<int>::I</code>
a member function declaration	<code>void A<int>::f(int, char)</code>
a template function declaration	<code>char* f(int, float)</code>

If instantiation pragmas are not used, the compiler will chose object files in which to instantiate all required instances automatically during the prelinking process.

#pragma instantiate *instance*

The `#pragma instantiate instance` requests the compiler to instantiate *instance* in the current compilation. For example,

```
#pragma instantiate class Stack<int>
```

will cause all static members and member functions for the `int` instance of a template class `Stack` to be instantiated, whether they are required in this compilation or not. The example,

```
#pragma instantiate void Stack<int>::push(int)
```

will cause only the individual member function `Stack<int>::push(int)` to be instantiated.

#pragma do_not_instantiate *instance*

The `#pragma do_not_instantiate instance` directs the compiler not to instantiate *instance* in the current compilation. For example,

```
#pragma do_not_instantiate int Stack<float>::use_count
```

will prevent the compiler from instantiating the static data member `Stack<float>::use_count` in the current compilation.

#pragma can_instantiate *instance*

The `#pragma can_instantiate instance` tells the compiler that if *instance* is required anywhere in the program, it should be instantiated in this compilation.



Currently, this pragma forces the instantiation even if it is not required anywhere in the program. Therefore, it has the same effect as `#pragma instantiate`.

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.

Increments and Decrements

Care should be taken when using increments or decrements in expressions. The C and C++ standards do not define the order that operations should take place in every circumstance.

For example, if in the following assignment expression,

```
a[i] = i++;
```

the intention might be to put the value of `i` into array `a` at index `i`, then increment `i` ready for assignment into the next element of `a`. However, it is up to the compiler when to perform the post-increment operation to make full use of the functionality available on the target machine. In such cases, the behavior of the code may not be as the user intended.

To be safe, it is recommended that you do not use an increment or decrement operator with a variable that appears more than once in a single expression. Instead, separate out the increment or decrement and put it in the desired place.

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;
```

and a C++ compiler and `ccts` process as:

```
a = b;
```

C++ Fractional Type Support

While in C++ mode, the `ccts` 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 32-bit fractional arithmetic is available on the TigerSHARC processors.

This section describes:

- [“Format of Fractional Literals” on page 1-169](#)
- [“Conversions Involving Fractional Values” on page 1-169](#)
- [“Fractional Arithmetic Operations” on page 1-169](#)
- [“Mixed Mode Operations” on page 1-170](#)

Fractional values are declared with the `fract` data type. Ensure that your program includes the `<fract>` header file. The `fract` data type 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 $[-1.0 \dots +1.0)$ 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.



Note that this implementation 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 `ccts` compiler validates fractional literal values 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. After the conversion of a 32-bit `double` literal to a `fract` literal, the value of the latter retains only 24 bits of precision compared with the full 32 bits for a fractional literal with the “r” suffix.

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 not 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 compiler:

- Standard arithmetic operations on two `fract` items include addition, subtraction, and multiplication.
- Assignment operations include `+=`, `-=`, and `*=`.

C/C++ Compiler Language Extensions

- 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.
- When arithmetic expressions contain a mixture of `fract` and `float` (or `double`) items, then the `float` (or `double`) items are normally converted to `fract` representation. For more information, see [“Mixed Mode Operations”](#).
- Multiplication of a fractional and an integer produces an integer result or a fractional result. The program context determines which type of result 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 must explicitly cast the result of the multiply operation if the error occurs.

Mixed Mode Operations

Most operations that are supported for fractional values are supported for mixed fractional/float or fractional/double arithmetic expressions. At runtime, 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.

GCC Compatibility Extensions

The compiler provides compatibility with the C dialect accepted by version 3.2 of the GNU C Compiler. Many of these features are available in the C99 ANSI Standard. A brief description of the extensions is included in this section. For more information, refer to the following web address:

<http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/C-Extensions.html#C%20Extensions>

Statement Expressions

A statement expression is a compound statement enclosed in parentheses. A compound statement itself is enclosed in braces { }, so this construct is enclosed in parentheses-brace pairs ({ }).

The value computed by a statement expression is the value of the last statement which should be an expression statement. The statement expression may be used where expressions of its result type may be used. But they are not allowed in constant expressions.

Statement expressions are useful in the definition of macros as they allow the declaration of variables local to the macro. In the following example,

```
#define min(a,b) ({
    short __x=(a),__y=(b),__res;
    if (__x > __y)
        __res = __y;
    else
        __res = __x;
    __res;
})

int use_min() {
    return min(foo(), thing()) + 2;
}
```

C/C++ Compiler Language Extensions

The `foo()` and `thing()` statements get called once each because they are assigned to the variables `__x` and `__y` which are local to the statement expression that `min` expands to and `min()` can be used freely within a larger expression because it expands to an expression.

Labels local to a statement expression can be declared with the `__label__` keyword. For example,

```
({
    __label__ exit;
    int i;
    for (i=0; p[i]; ++i) {
        int d = get(p[i]);
        if (!check(d)) goto exit;
        process(d);
    }
    exit:
    tot;
})
```



Statement expressions are not supported in C++ mode.



Statement expressions are an extension to C originally implemented in the GCC compiler. Analog Devices support the extension primarily to aid porting code written for that compiler. When writing new code, consider using inline functions, which are compatible with ANSI/ISO standard C++ and C99, and are as efficient as macros when optimization is enabled.

Type Reference Support Keyword (`typeof`)

The `typeof( expression )` construct can be used as a name for the type of expression without actually knowing what that type is. It is useful for making source code that is interpreted more than once such as macros or include files more generic.

The `typeof` keyword may be used where ever a `typedef` name is permitted such as in declarations and in casts. For example,

```
#define abs(a) ({
    typeof(a) __a = a;
    if (__a < 0) __a = - __a;
    __a;
})
```

shows `typeof` used in conjunction with a statement expression to define a “generic” macro with a local variable declaration.

The argument to `typeof` may also be a type name. Because `typeof` itself is a type name, it may be used in another `typeof( type-name )` construct. This can be used to restructure the C type declaration syntax.

For example,

```
#define pointer(T)    typeof(T *)
#define array(T, N)  typeof(T [N])

array (pointer (char), 4) y;
```

declares `y` to be an array of four pointers to `char`.



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



The `typeof` keyword is an extension to C originally implemented in the GCC compiler. It should be used with caution because it is not compatible with other dialects of C or C++ and has not been adopted by the more recent C99 standard.

GCC Generalized Lvalues

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

C/C++ Compiler Language Extensions

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

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

Conditional Expressions with Missing Operands

The middle operand of a conditional operator can be left out. If the condition is non-zero (true), then the condition itself is the result of the expression. This can be used for testing and substituting a different value when a pointer is NULL. The condition is only evaluated once; therefore, repeated side effects can be avoided. For example,

```
printf("name = %s\n", lookup(key)?:"-");
```

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

Hexadecimal Floating-Point Numbers

C99 style hexadecimal floating-point constants are accepted. They have the following syntax.

```
hexadecimal-floating-constant:  
    {0x|0X} hex-significand binary-exponent-part [ floating-suffix ]  
hex-significand: hex-digits [ . [ hex-digits ] ]  
binary-exponent-part: {p|P} [+|-] decimal-digits  
floating-suffix: { f | l | F | L }
```

The hex-significand is interpreted as a hexadecimal rational number. The digit sequence in the exponent part is interpreted as a decimal integer. The exponent indicates the power of two by which the significand is to be scaled. The floating suffix has the same meaning that it has for decimal

floating constants: a constant with no suffix is of type `double`, a constant with suffix `F` is of type `float`, and a constant with suffix `L` is of type `long double`.

Hexadecimal floating constants enable the programmer to specify the exact bit pattern required for a floating-point constant. For example, the declaration

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

causes `f` to be initialized with the value `0x800000`.

 Hexadecimal floating constants are not supported in C++ mode.

Zero Length Arrays

Arrays may be declared with zero length. This is an anachronism supported to provide compatibility with GCC. Use variable length array members instead.

Variable Argument Macros

The final parameter in a macro declaration may be followed by `...` to indicate the parameter stands for a variable number of arguments.

For example,

```
#define trace(msg, args...) fprintf (stderr, msg, ## args);
```

can be used with differing numbers of arguments,

```
trace("got here\n");
trace("i = %d\n", i);
trace("x = %f, y = %f\n", x, y);
```

C/C++ Compiler Language Extensions

The `##` operator has a special meaning when used in a macro definition before the parameter that expands the variable number of arguments: if the parameter expands to nothing, then it removes the preceding comma.



The variable argument macro syntax comes from GCC. It is not compatible with C99 variable argument macros and is not supported in C++ mode.

Line Breaks in String Literals

String literals may span many lines. The line breaks do not need to be escaped in any way. They are replaced by the character `\n` in the generated string. This extension is not supported in C++ mode. The extension is not compatible with many dialects of C, including ANSI/ISO C89 and C99. However, it is useful in `asm` statements, which are intrinsically non-portable.

Arithmetic on Pointers to Void and Pointers to Functions

Addition and subtraction is allowed on pointers to `void` and pointers to functions. The result is as if the operands had been cast to pointers to `char`. The `sizeof()` operator returns one for `void` and function types.

Cast to Union

A type cast can be used to create a value of a union type, by casting a value of one of the unions member's types.

Ranges in Case Labels

A consecutive range of values can be specified in a single case by separating the first and last values of the range with `...`.

For example,

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

Declarations Mixed with Code

In C mode, the compiler will accept declarations in the middle of code as in C99 and C++. This allows the declaration of local variables to be placed at the point where they are required. Therefore, the declaration can be combined with initialization of the variable.

For example, in the following function

```
void func(Key k) {
    Node *p = list;
    while (p && p->key != k)
        p = p->next;
    if (!p)
        return;
    Data *d = p->data;
    while (*d)
        process(*d++);
}
```

the declaration of `d` is delayed until its initial value is available, so that no variable is uninitialized at any point in the function.

Escape Character Constant

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

Alignment Inquiry Keyword (`__alignof__`)

The `__alignof__` (type-name) construct evaluates to the alignment required for an object of a type. The `__alignof__ expression` construct can also be used to give the alignment required for an object of the *expression type*.

If *expression* is an lvalue (may appear on the left hand side of an assignment), the alignment returned takes into account alignment requested by pragmas and the default variable allocation rules.

Keyword for Specifying Names in Generated Assembler (asm)

The `asm` keyword can be used to direct the compiler to use a different name for a global variable or function (see also “[#pragma linkage_name identifier](#)” on page 1-163). For example,

```
int N asm("C11045");
```

tells the compiler to use the label C11045 in the assembly code it generates wherever it needs to access the source level variable N. By default, the compiler would use the label `_N`.

The `asm` keyword can also be used in function declarations but not function definitions. However, a definition preceded by a declaration has the desired effect. For example,

```
extern int f(int, int) asm("func");

int f(int a, int b) {
    . . .
}
```

Function, Variable and Type Attribute Keyword (`__attribute__`)

The `__attribute__` keyword can be used to specify attributes of functions, variables and types, as in these examples,

```
void func(void) __attribute__((section("fred")));
int a __attribute__((aligned (8)));
typedef struct {int a[4];} __attribute__((aligned (4))) Q;
```

The `__attribute__` keyword is supported, and therefore code, written for GCC, can be ported. All attributes accepted by GCC on ix86 are accepted. The ones that are actually interpreted by the `ccts` compiler are described in the sections of this manual describing the corresponding pragmas (see “[Pragmas](#)” on page 1-148).

Preprocessor Features

The `ccts` 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” on page 1-167](#).

This section describes:

- [“Predefined Preprocessor Macros”](#)
- [“Header Files” on page 1-181](#)
- [“Writing Macros” on page 1-182](#)
- [“Preprocessing of .IDL Files” on page 1-184](#)

Predefined Preprocessor Macros

The `ccts` 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:

Preprocessor Features

<code>__ADSPTS__</code>	<code>ccts</code> always defines <code>__ADSPTS__</code> as 1. This indicates that a TigerSHARC DSP program is being compiled.
<code>__ADSPTS101__</code>	<code>ccts</code> defines <code>__ADSPTS101__</code> as 1 when you compile with the <code>-proc ADSP-TS101</code> command-line switch.
<code>__ADSPTS201__</code>	<code>ccts</code> defines <code>__ADSPTS201__</code> as 1 when you compile with the <code>-proc ADSP-TS201</code> command-line switch.
<code>__ADSPTS202__</code>	<code>ccts</code> defines <code>__ADSPTS202__</code> as 1 when you compile with the <code>-proc ADSP-TS202</code> command-line switch.
<code>__ADSPTS203__</code>	<code>ccts</code> defines <code>__ADSPTS203__</code> as 1 when you compile with the <code>-proc ADSP-TS203</code> command-line switch.
<code>__ADSPTS20x__</code>	<code>ccts</code> defines <code>__ADSPTS20x__</code> as 1 when you compile with either the <code>-proc ADSP-TS201</code> , <code>-proc ADSP-TS202</code> , or <code>-proc ADSP-TS203</code> command-line switch.
<code>__ANALOG_EXTENSIONS__</code>	<code>ccts</code> defines <code>__ANALOG_EXTENSIONS__</code> as 1, unless you compile with <code>-pedantic</code> or <code>-pedantic-errors</code> .
<code>__cplusplus</code>	<code>ccts</code> defines <code>__cplusplus</code> as 1 when you compile in C++ mode.
<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>ccts</code> always defines <code>__DOUBLES_ARE_FLOATS__</code> as 1.
<code>__ECC__</code>	<code>ccts</code> always defines <code>__ECC__</code> as 1.
<code>__EDG__</code>	<code>ccts</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>ccts</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>ccts</code> command line or in a preprocessor <code>#include</code> command (ANSI standard).
<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>ccts</code> defines <code>__NO_BUILTIN</code> as 1 when you compile with the <code>-no-builtin</code> command-line switch.

<code>__SIGNED_CHARS__</code>	<code>ccts</code> defines <code>__SIGNED_CHARS__</code> as 1 unless you compile with the <code>-unsigned-char</code> command-line switch.
<code>__STDC__</code>	<code>ccts</code> always defines <code>__STDC__</code> as 1.
<code>__STDC_VERSION__</code>	<code>ccts</code> always defines <code>__STDC_VERSION__</code> as 199409L.
<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).
<code>__TS_BYTE_ADDRESS</code>	<code>ccts</code> defines this macro if byte-addressing mode is selected using the <code>-char-size-8</code> switch.
<code>__SILICON_REVISION__</code>	The <code>__SILICON_REVISION__</code> macro is defined when the <code>-si-revision</code> switch is specified with a value other than “none”. The value it is defined to is the major revision number left shifted by 8 logically or’d against the minor revision number.
<code>__VERSION__</code>	The preprocessor expands this macro into a string constant containing the current compiler version.
<code>__WORKAROUNDS_ENABLED</code>	<code>ccts</code> defines this macro to be 1 if any hardware workarounds are implemented by the compiler. This macro is set if the <code>-si-revision</code> switch has a value other than “none” or if any specific workaround is selected by means of the <code>-workaround</code> switch.

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 an `.h` extension. There are two main categories of header files:

- System header files declare the interfaces to the libraries supplied with the TigerSHARC 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); \
    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](#).

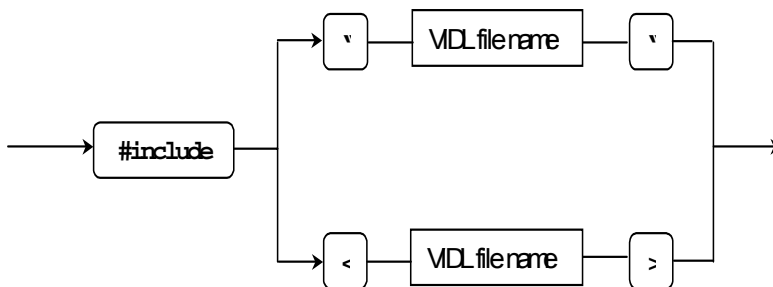


Figure 1-2. `#include` Syntax Diagram

The file identified by the file name is located by searching a list of directories. When the name is delimited by quote characters, the search begins in the directory containing the primary input file, then proceeds with the list of directories specified by the `-I` command-line switch. When the name is delimited by angle-bracket characters, the search proceeds directly with the directories specified by `-I`. If the file is not located within any directory on the search list, the search may be continued in one or more platform dependent system directories.

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 TigerSHARC 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 TigerSHARC processors. Assembly routines linked to C/C++ routines must follow these conventions.



ADI recommends that assembly programmers maintain stack conventions.

This section describes:

- [“Stack Frame Overview”](#)
- [“Stack Frame Description” on page 1-188](#)
- [“Using Multiple Heaps” on page 1-200](#)
- [“Miscellaneous Information” on page 1-204](#)
- [“Register Classification” on page 1-204](#)

Stack Frame Overview

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.

C/C++ Run-Time Model and Environment

- 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. Each row is a quad word group; the higher addresses are on the left since this is a little-endian machine.

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

Figure 1-3. ADSP-TS101 DSP Stack

There are two stacks and thus two current stack frames at any point on the TigerSHARC processors. One stack resides in each of the data memory banks. One stack is addressed from the j i ALU registers, the other stack is addressed from the k register. For more information, refer to “[Stack Frame Description](#)” on page 1-188.

Each stack 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.



The second stack is not yet used for local variables.

The following design attributes have been included in the TigerSHARC processor’s run-time model for additional efficiency:

- The frame pointers are offset by -0×40 from the actual base of the frame. This provides greater addressing range for larger negative offsets (for local variables) than positive ones (for arguments). Since all FP-based references are relative (for example, with an offset), there is no run-time cost and little extra complexity. This offset is constant throughout the system; it does not change for particular routines. However, this may cause the Frame Pointer address to actually be off the end of the stack.
- Optionally, a copy of the j frame pointer may be kept in register k_{24} . This increases performance by allowing simple j FP-relative memory references to occur in the same instruction line as other [address] calculations in the j ALU. This copy is made at the discretion of the particular procedure. A similar copy of k_{SP} into j is also possible.
- A routine that does not wish to use the k stack need not do so. If it does not use the k stack, it can dispense with all operations, including linkage related to the k stack and k stack pointers.



- At present, a `k` frame must always be allocated in order for the debugger to be able to walk back up the stack. At some future point, it is expected that the debug information will be enriched to include information on whether or not a `k` frame is present.
- Space is allocated in the caller's frame for at least four "argument words" (see in ["Stack Frame Description"](#)). For routines with few arguments, this space is available for use as temporary storage. That may be of use for small, leaf routines, which may be able to avoid having to create a stack frame.

Stack Frame Description

This section describes the stack as shown in [Figure 1-3 on page 1-186](#).

Incoming Arguments

The memory area for incoming arguments begins at the effective `jFP` value +8, or actual `j26 + 0x48`. Argument words are mapped by ascending addresses, so the second argument word is mapped at `j26+0x49`. Note that the first four words arrive in registers, although space is allocated in memory as well.

Linkage Information

The return address is saved in the `CJMP` register by the `CALL` instruction. This is saved at `jSP+0 (j27+0)` on entry. It is retrieved from effective `jFP + 0 (j26+64)` at exit.

Local Variables and Temporaries

Space for a register save area and local variables/temporaries is allocated on both stacks by the procedure prologue. The save area begins after the outgoing argument space, typically at `jSP+12` and `kSP+8`. This can also be addressed from `jFP` and `kFP`. Local variable/temp storage begins after the save area. It is addressed at

effective $j_{FP}\text{-offset}$ or effective $k_{SP}\text{-offset}$. Since the actual FP registers are offset by a constant 0×40 , the actual offsets might be positive for some cases. This should cause no problems.

Outgoing Arguments

Space for outgoing arguments is allocated on the stack at $j_{SP}+8$ ($j_{27}+8$). Four words are always available as part of the basic stack frame. If more are needed, they can either be allocated as part of the prologue or by local extension of the stack frame for the purpose of a particular call.

Outgoing Linkage

Four words, at $j_{SP}+4$ ($j_{27}+4$) and $k_{SP}+4$ ($k_{27}+4$), are available immediately to the callee for saving linkage information. These words are designated for holding the quad registers containing the j and k stack pointer registers. Four words, at $j_{SP}+0$ ($j_{27}+0$) and $k_{SP}+0$ ($k_{27}+0$), are also available at any time, and may be used by the callee.

Free Space

Space below j_{SP} and k_{SP} is considered free and unprotected. It is available for use (in growing the stack) at any time, synchronously or asynchronously (the latter for interrupt handling). This space should never be used.

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 value in FP plus 0×40).

- The first free quad-word in each stack is addressed by the `SP` register. Locations at that point and beyond are vulnerable and must not be used. These locations may be clobbered by asynchronous activities, such as interrupt service routines. Alternatively, locations at `SP` and beyond are always available if additional space is needed, but `SP` must be moved to guard the space. Therefore, the first “used” (protected) word is at `SP+4`.



Data can be pushed onto the stack by executing an instruction like `[jSP += -4] = reg;` for either `j` or `k`.

- The address of each `FP` must always be four-word aligned; that is, the low-order two address bits are always zero.
- Likewise, the `SP` registers should also be kept four-word aligned at all times. This allows interrupt routines to save registers without having to first verify stack alignment.
- The return address of the caller is stored at offset zero from the address carried by the current effective `jFP`.
- The linkage back to the previous stack frame is stored at offset +6 and +7 from each current effective `FP`.

At a procedure call, the following must be true:

- Each `SP` must be four-word aligned (as it will form the basis for the new frame’s `FP`).
- There must be eight words available starting at `jSP+4` and four words available starting at `kSP+4`. The first four words on each stack are used for storing linkage information, and the remaining four (and perhaps others) on the `j` stack hold arguments. There is always space allocated for at least four arguments, even if the current procedure does not have that many. That way, the callee can always store the argument registers. Small routines are free to use unneeded argument slots for local storage.

- The standard calling convention further specifies that the previous frame's linkage information will already be stored in the standard slots by the time control reaches the new procedure. However, the callee is responsible for restoring this information prior to exit.

The saving of the frame linkage can be performed as part of the instruction line containing the procedure call. In some instances, this information will remain constant for several outgoing calls, in which case it need not be repeatedly stored.

Argument Passage

A contiguous block of memory, near the end of the current frame, is provided to hold all arguments. Argument handling begins by conceptually “mapping” all the arguments into this area. The arguments are laid out into ascending addresses. They must obey alignment constraints based on their size. An argument might occupy more than one word; there might also be a vacant word, or “hole”, in the argument area to maintain alignment. This is the definition of “argument words”.

Data corresponding to the first four words of the argument area are normally passed in registers, rather than on the stack, which increases efficiency. For a common situation where each argument occupies a single word, the first four arguments can be passed in registers. There is also space allocated on the stack so that these register values can be stored back in the prologue in their canonical places.

The choice of register file used for argument passage is based on the argument's declared type. Although two register files are available, only four argument words are passed. The corresponding word in the other register file is unused. (See [Table 1-14 on page 1-192](#)).

Rules governing argument words are:

- Pointers and integral types of one word are passed in the *j* registers.
- Floating types, integral types of more than one word, and structure (or unions) that fit, are passed in the *x* registers. This use of registers to pass structures or unions operates independently of the type of the structure or union fields. (A structure composed entirely of pointers is still passed in *x*). The alignment constraints are still observed.
- If an argument requires sufficient space that causes it to span over the end of the register argument area (typically, if it requires more than two words), it and all succeeding arguments are passed in memory in the usual place rather than in the registers.
- If the prototype specifies `varargs` (“...”), then the argument immediately preceding the ellipsis and all succeeding arguments are passed in memory. Note that `varargs` routines **MUST** have prototypes in order to function correctly.
- If there is no prototype, the actual types of the arguments are used to determine how they are passed.
- Large structures may be passed by value. The structures are copied into the argument area, just as with other arguments.

Table 1-14. Register–Argument Word Correspondence

Argument Word	Register Used	
	if pointer or int	if float or double word
Arg word 1	j4	xR4
Arg word 2	j5	xR5
Arg word 3	j6	xR6
Arg word 4	j7	xR7

Return Values

Return values always use registers. The type of the value determines, as with arguments, whether to use the *j* or *x* register file (*j*8, or *x*R8 and *x*R9).



Two *x* return registers are identified to accommodate double-word result values.

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 *j* return register, *j*8, is used for this purpose. The procedure must also return the [same] pointer value in that register on return. This is an optimization to allow more efficient referencing of the returned value.

Procedure Call and Return

The following steps describe how to manage procedure calls and returns.

To Call a Procedure:

1. Expand the current stack frame, if necessary, to provide space for the outgoing arguments and linkage information.

The number of arguments must be rounded up to a multiple of four in order to maintain proper alignment.

It is recommended that you establish and retain a basic call area during prologue. This saves you from creating and removing the area at each procedure call. If the number of arguments is large or a nested call occurs you must do a full expand-remove cycle.

2. Evaluate the arguments and set them up in the argument area and/or registers.

If another function must be called as part of computing an argument, then the stack can be extended (temporarily) a second time. If you extend the stack a second time, it's probably best to store even the register arguments of the outer procedure into the stack (for safekeeping), then load them just prior to the actual call.

3. Call the procedure, saving the current frame pointers in the appointed slots.
4. On return, remove the frame expansion if desired.

On Entry:

1. Set the new frame pointers (in both iALU's) to the value of the current stack pointer, less the 0×40 offset. This can be done simultaneously in both iALU's.
2. Set up a new frame and continue saving context: store the `cjmp` (return) register on the `j` stack at offset +0 from the current (old) stack pointer. By using post-modify addressing on the store instruction, the stack pointer may simultaneously be moved down to create a new frame.

Registers are saved on the `k` stack as needed while the frame is created.



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

3. If debug is specified (`-g` switch), arguments arriving in registers should be stored back into memory so the debugger can find them.



This restriction may be lifted in future releases when the debug tables are more expressive.

4. If desired, transfer a copy of the new `jFP` register to `k`.
5. Continue saving registers and then executing the procedure.

A leaf procedure that does not require much stack space might choose to omit steps (1) and (2), operating without its own stack frame. A small amount of local storage is available in any unused argument slots since there are always at least four slots as well as the two quad words reserved for saving linkage information.

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

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

To Return from a Procedure:

1. If a call was made to another procedure, then `CJMP` must be restored.

For best performance, restore `CJMP` as soon as possible after the last procedure call or computed jump, (such as a switch). If the return jump is predicted, stalls can be avoided provided that `CJMP` is reloaded four cycles plus twelve (12) words prior to the return.

2. Restore miscellaneous saved registers.
3. Place the return value in the correct register (if not there already).
4. Restore the stack pointers for the previous frame. This can be accomplished, for each stack, with a single quad load.
5. Return to the caller.

Code Sequences

The following example shows the normal code sequences used for the various actions involved in calling a procedure and returning from a procedure.

```
// Extend stack frame if necessary (more than 4 arguments)
jsp = jsp - nn; ksp = ksp - mm;;      // must be multiple of 4
```

C/C++ Run-Time Model and Environment

```
// Evaluate arguments. Store into memory, or into arg registers.  
...  
// Now do the call.  
call subr; q[jSP+4]=j27:24; q[kSP+4]=k27:24;;
```

The fourth slot might be available for some computation or it may be used by the IMEX forming the full address for the call.

```
// if the stack was extended, cut it back.  
jSP = jSP + nn; kSP = kSP + mm;; // must be multiple of 4
```

Prologue:

```
jFP = jSP - 0x40; kFP = kSP - 0x40;;  
[jSP += -nnj]=CJMP; q[kSP += -nnk]=<some regs>;  
// At this point, the new frame is intact.  
q[jSP+n] = <some quad>; kjFPcopy = jFP;
```

Epilogue:

```
// Restore CJMP after last outgoing call  
CJMP = [jFP+0x40];
```

More code:

```
// Nearing end, restore other saved registers.  
...  
// Exit: Restore stack linkage and return.  
jump CJMP; j27:24 = q[jFP+0x44]; k27:24=q[kFP+0x44];;
```



There is a four-cycle wait before either frame pointer can be used again. If the return jump is not predicted correctly, the four-cycle wait will cover the delay. Otherwise, callers might want to refrain from immediately accessing the frame pointers on return.

You should not reset the stack pointers prior to return. That would create a time window in which the stack has been reset to a frame not corresponding to the current procedure.

Support for argv/argc

By default, the facility to specify arguments that get passed to your `main()` (`argv/argc`) at run-time is not enabled. To enable it, you need to modify application builds in the following ways:

1. Define your command-line arguments in C by defining a variable called “`__argv_string`”. When linked, your new definition will over-ride the default zero definition otherwise found in the C run-time library. For example,

```
const char __argv_string[] = "-in x.gif -out y.jpeg";
```

2. To use command-line arguments as part of Profile-Guided Optimizations (PGO), it is necessary to define `__argv_string` within a memory section called `SEG_ARGV`. Therefore, define a memory section called `seg_argv` in your `.LDF` file and include the definition of `__argv_string` in it if using PGO. The default `.LDF` files will do this for you if macro `IDDE_ARGS` is defined at link time.

File I/O Support

The VisualDSP++ environment provides access to files on a host system by using `stdio` functions. File I/O support is provided through a set of low-level primitives that implement the `open`, `close`, `read`, `write`, and `seek` operations. The `stdio` functions make use of these primitives to provide conventional C input and output facilities. The source files for the I/O primitives are available under the VisualDSP++ installation in the sub-directory `TS\lib\src\libc`.

Refer to “[stdio.h](#)” on page 3-14 for more information.

Extending I/O Support To New Devices

The I/O primitives are implemented using an extensible device driver mechanism. The default startup code includes a device driver that can perform I/O through the VisualDSP++ simulator and EZ-KIT Lite evaluation systems. Other device drivers may be registered and then used through the normal `stdio` functions.

A device driver is a set of primitive functions grouped together into a `DevEntry` structure. This structure is defined in `device.h`:

```
struct DevEntry {
    int    DeviceID;
    void   *data;

    int     (*init)(struct DevEntry *entry);
    int     (*open)(const char *name, int mode);
    int     (*close)(int fd);
    int     (*write)(int fd, unsigned char *buf, int size);
    int     (*read)(int fd, unsigned char *buf, int size);
    long    (*seek)(long fd, long offset, int whence);
};

typedef struct DevEntry DevEntry;
typedef struct DevEntry *DevEntry_t;
```

The `DeviceID` field is a unique identifier for the device, known to the user. Device IDs are used globally across an application. The `data` field is a pointer for any private data the device may need; it is not used by the run-time libraries. The function pointed to by the `init` field is invoked by the run-time library when the device is first registered. It returns a negative value for failure or a positive value for success.

The functions pointed to by the `open`, `close`, `write` and `read` fields are the functions that provide the same functionality used in the default I/O device. `Seek` is another function at the same level, for those devices which support such functionality. If a device does not support an operation (such

as seeking, writing on read-only devices or reading write-only devices), then a function pointer must still be provided; the function must arrange to always return failure codes when the operation is attempted.

A new device can be registered with the following call:

```
int add_devtab_entry(DevEntry_t entry);
```

If the device is successfully registered, the `init()` routine of the device is called, with `entry` as its parameter. `add_devtab_entry()` returns the DeviceID of the device registered.

If the device is not successfully registered, a negative value is returned. Reasons for failure include:

- The DeviceID is the same as another device, already registered
- There are no more spaces left in the device registry table
- The DeviceID is less than zero
- Some of the function pointers are NULL
- The device's `init()` routine returned a failure result

Once a device is registered, it can be made the default device, using the following function:

```
void set_default_io_device(int);
```

The user passes the DeviceID. There is a corresponding function for retrieving the current default device:

```
int get_default_io_device(void);
```

The default device is used by `fopen()` when a file is first opened. The `fopen()` function passes the open request to the `open()` function of the device indicated by `get_default_io_device()`. The device file identifier (`dfid`) returned by the `open()` function is private to the device; other

devices may simultaneously have other open files that use the same identifier. An open file is uniquely identified by the combination of `DeviceID` and `dfid`.

The `fopen()` function records the `DeviceID` and `dfid` in the global open file table, and allocates its own internal `fid` to this combination. All future operations on the file reads, writes, seeks and close—use this `fid` to retrieve the `DeviceID`—and thus direct the request to the appropriate device's primitive functions, passing the `dfid` along with other parameters. Once a file has been opened by `fopen()`, the current value of `get_default_io_device()` is irrelevant to that file.

Using Multiple Heaps

The TigerSHARC C/C++ run-time library supports the standard heap management functions `calloc`, `free`, `malloc`, and `realloc`. By default, these functions access the default heap, which is defined in the standard Linker Description File and the run-time header.

User written code can define any number of additional heaps. These additional heaps can be accessed either by the standard `calloc`, `free`, `malloc` and `realloc` functions, or via the extension routines `heap_calloc`, `heap_free`, `heap_malloc` and `heap_realloc`.

Each heap must be declared in the heap table in the `.LDF` file and the `TS\lib\src\crt_src\ts_hdr.asm` file must declare memory and section placement for the heaps. Refer to the example below on how to modify the header object and `.LDF` file. The default `ts_hdr.asm` file declares one default heap. To use a custom `ts_hdr.asm`, assemble and use it to replace the default `ts_hdr_TS101.doj` that is specified in your copy of the `.LDF` file. The calculation for a heap's size and length occur in the project's linker description file. When linking, the linker handles substitution of values to resolve the heap's definition (the `.VAR` directive in the `ts_hdr.asm` file).

Heap Identifiers

The primary heap ID is the index of the descriptor for that heap in `ts_hdr.asm`. The default heap, `seg_heap`, is always 0 and the primary heap IDs of user defined heaps will be 1, 2, 3 and so on.

Example:

The following `my.ldf` extract contains the default heap and alternate heap entries.

```
// The default heap occupies its own memory block defheapseg
{
    ldf_defheap_base = .;
    ldf_defheap_size = MEMORY_SIZEOF(M1Heap);
} >M1Heap
altheapseg
{
    ldf_altheap_base=.;
    ldf_altheap_size = MEMORY_SIZEOF(SDRAM);
} >SDRAM
```



Note that `ldf_defheap_base` **and** `ldf_defheap_size` **are set up for use in the heap table setup.**

The following `ts_hdr.asm` is an extract which contains the default heap entry and one additional alternate heap.

```
// Create the heap descriptor table and describe the default
// heap, which is the first entry in the heap descriptor
// table. The ts_exit.asm file declares a label for the end of
// this table.

.SECTION heaptab;
.GLOBAL __heaptab_start;
__heaptab_start:
.VAR = ldf_defheap_base; // Start of default heap
.VAR = ldf_defheap_size; // Size of default heap,
                        // unit==sizeof(char)
.VAR = 0;               // ID of default heap - must be 0
```

C/C++ Run-Time Model and Environment

```
.VAR = ldf_altheap_base; // Start of alt heap
.VAR = ldf_altheap_size; // Size of alt heap, unit==sizeof(char)
.VAR = 1;                // Increment ID of alt heap must not be 0
```

Initializing the Heap

The default heap is initialized by the run-time library. Additional heap can be initialized by a call to `heap_init` which takes the heap ID as its parameter which would normally be done at the beginning of `main`.

Using Alternate Heaps with the Standard Interface

Alternate heaps can be accessed by the standard functions `calloc`, `free`, `malloc` and `realloc`. The run-time library keeps track of the current heap, which initially is the default heap. The current heap can be changed any number of times at run time by calling `heap_switch` with the heap ID as a parameter. The standard functions `calloc` and `malloc` always allocate a new object from the current heap. If `realloc` is called with a null pointer, it too will allocate a new object from the current heap.

Previously allocated objects can be deallocated with `free` or `realloc`, or resized with `realloc`, even if the current heap is now different from when the object was originally allocated. When a previously allocated object is resized with `realloc`, the returned object will always be in the same heap as the original object.

Using the Alternate Heap Interface

The C run-time library provides the alternate heap interface functions `heap_free`, `heap_calloc`, `heap_malloc`, and `heap_realloc`. These routines work exactly the same as the corresponding standard functions without the “heap_” prefix, except that they take an additional argument that specifies the heap ID. These functions are completely independent of the current heap setting.

Objects allocated with the alternate interface functions can be freed with either the `free` or `heap_free` (or `realloc` or `heap_realloc`) functions. The `heap_free` function is a little faster than `free` since it doesn't have to search for the proper heap. However, it is essential that the `heap_free` or `heap_realloc` functions be called with the same heap ID that was used to allocate the object being freed. If it is called with the wrong heap ID, the object would not be freed or reallocated.



The `heap_switch()` interface is not available when using the multi-threaded run-time support. However, all of the other alternate heap interface functions are available.

The actual entry point names for the alternate heap interface routines have an initial underscore, that is, they are `_heap_switch`, `_heap_calloc`, `_heap_free`, `_heap_malloc`, and `_heap_realloc`. The `stdlib.h` standard header file defines macro equivalents without the leading underscores.

Example C Program

```
// Example program using the standard heap interface
// along with an additional heap as specified in the
// previous example

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

main()
{
    int *w, *x, *y, *z;
    heap_init(1);           // Initailiase the alternate heap
    w = malloc(1000);       // Get 1K words of default heap space
    heap_switch(1);         // Set the current heap to the alternate
    heap x = malloc(1000);  // Get 1K words of alternate heap space
                           // in case it is referred to elsewhere
    y = heap_malloc(1, 1000); // By specifying the alternate heap
                           // allocate 1K words on the alternate heap
    z = malloc(1000);       // Allocate 1K words on the current
                           // (default) heap
}
```

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. Since pointers and `ints` are handled in the same way, some common interface errors are tolerated. Mixing up `float` and `integer` arguments generally results in incorrect behavior.
- There is no special interface for calling system library functions. They use the standard calling convention.
- Procedures that use the `stdargs` (`varargs`) mechanism need to have prototypes.
- The `k` stack has a slightly irregular existence at present. There must always be a frame, but it cannot be used for local variables. These restrictions are both based on limitations in the debug information.

Register Classification

This section describes all of the TigerSHARC processor registers. Registers are listed in order of preferred allocation by the compiler.

Callee Preserved Registers (“Preserved”)

Registers `j16` through `j25`, `k16` through `k25`, `x24` through `x31`, `y24` through `y31` are “preserved” (or dedicated). A subroutine that uses any of these registers must save (preserve) it and restore it.

Caller Save Registers (“Scratch”)

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

ADSP-TS101 and ADSP-TS20x Processor Registers

The following is a list of registers for the ADSP-TS101 processors and ADSP-TS20x processors. The registers are listed in order of preferred allocation by the compiler.

- [Table 1-15 on page 1-206](#) — IALU (j & k) General Registers
- [Table 1-16 on page 1-207](#) — Compute Block General Registers
- [Table 1-17 on page 1-208](#) — IALU (j & k) Special Registers
- [Table 1-18 on page 1-208](#) — Compute Block X MAC Registers
- [Table 1-19 on page 1-209](#) — Compute Block Y MAC Registers
- [Table 1-20 on page 1-209](#) — Compute Block ALU Summation Registers
- [Table 1-21 on page 1-209](#) — Loop Counters
- [Table 1-22 on page 1-209](#) — Data Alignment Registers
- [Table 1-23 on page 1-209](#) — Bit FIFO Temporaries Registers
- [Table 1-24 on page 1-209](#) — Static Condition Flags Registers
- [Table 1-25 on page 1-210](#) — Return Address Registers
- [Table 1-26 on page 1-210](#) — Enhanced Communications Registers (ADSP-TS101 processors only)
- [Table 1-27 on page 1-211](#) — Enhanced Communications Registers (ADSP-TS201 processors only)

C/C++ Run-Time Model and Environment

Table 1-15. iALU (j and k) General Registers

j31:	j30:	j29:	j28:	k31:	k30:	k29:	k28:
zero ¹	scratch	scratch	scratch	zero1	scratch	scratch	scratch

j27:	j26:	j25:	j24:	k27:	k26:	k2:	k2:
Dedicated jSP	Dedicated jFP	RES (or GOT)	RES	Dedicated kSP	Dedi- cated kFP	RES (or GOT)	RES kjFP Copy

j23:	j22:	j21:	j20:	k23:	k22:	k21:	k20:
RES	RES	RES	RES	RES	RES	RES	RES

j19:	j18:	j17:	j16:	k19:	k18:	k17:	k16:
RES	RES	RES	RES	RES	RES	RES	RES

j15:	j14:	j13:	j12:	k15:	k14:	k13:	k12:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

Table 1-15. iALU (j and k) General Registers (Cont'd)

j11:	j10:	j9:	j8:	k11:	k10:	k9:	k8:
scratch	scratch	scratch	scratch RTN0 ²	scratch	scratch	scratch	scratch

j7:	j6:	j5:	j4:	k7:	k6:	k5:	k4:
scratch ARG4	scratch ARG3	scratch ARG2	scratch ARG1	scratch	scratch	scratch	scratch

j3:	j2:	j1:	j0:	k3:	k2:	k1:	k0:
scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)	scratch (circ)

- 1 j/k 31 are zero registers when used for addressing, or as operands to iALU operations; but when used in a store or load (or ureg transfer), they become the j/k status register.
- 2 j8 is also used for passing the address of a hidden argument.

Table 1-16. Compute Block (x & y) General Registers

x31:	x30:	x29:	x28:	y31:	y30:	y29:	y28:
RES	RES	RES	RES	RES	RES	RES	RES

x27:	x26:	x25:	x24:	y27:	y26:	y25:	y24:
RES	RES	RES	RES	RES	RES	RES	RES

x23:	x22:	x21:	x20:	y23:	y22:	y21:	y20:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch
x19:	x18:	x17:	x16:	y19:	y18:	y17:	y16:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

Table 1-16. Compute Block (x & y) General Registers (Cont'd)

x15:	x14:	x13:	x12:	y15:	y14:	y13:	y12:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

x11:	x10:	x9:	x8:	y11:	y10:	y9:	y8:
scratch	scratch	scratch RTN 1	scratch RTN 0	scratch	scratch	scratch	scratch

x7:	x6:	x5:	x4:	y7:	y6:	y5:	y4:
scratch ARG4	scratch ARG3	scratch ARG2	scratch ARG1	scratch ARGn	scratch ARGn	scratch ARGn	scratch ARGn

x3:	x2:	x1:	x0:	y3:	y2:	y1:	y0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

Table 1-17. iALU (j & k) Special Registers: Circular Buffering – B (base) and L (length)

jB3:	jB2:	jB1:	jB0:	kB3:	kB2:	kB1:	kB0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

jL3:	jL2:	jL1:	jL0:	kL3:	kL2:	kL1:	kL0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

Table 1-18. Compute Block X MAC Registers

xMR4:		xMR3:	xMR2:	xMR1:	xMR0:
scratch		scratch	scratch	scratch	scratch

Table 1-19. Compute Block Y MAC Registers

yMR4:		yMR3:	yMR2:	yMR1:	yMR0:
scratch		scratch	scratch	scratch	scratch

Table 1-20. Compute Block ALU Summation Registers

xPR1:	xPR0:		yPR1:	yPR0:
scratch	scratch		scratch	scratch

Table 1-21. Loop Counters

LC1:		LC0:
scratch		scratch

Table 1-22. Data Alignment Registers

xDAB:		yDAB:
scratch		scratch

Table 1-23. Bit FIFO Temporaries Registers

xBFOTMP:		yBFOTMP:
scratch		scratch

Table 1-24. Static Condition Flags Registers

iSF0:	iSF1:		xF0:	xF1:		ySF0:	ySF1:
scratch	scratch		scratch	scratch		scratch	scratch

Table 1-25. Return Address Registers

CJMP:		RETI:
scratch – for caller reserved – by callee		Not Available

Table 1-26. Enhanced Communications Registers (ADSP-TS101 Processors only)¹

xtr15:	xtr14:	xtr13:	xtr12:	ytr15:	ytr14:	ytr13:	ytr12:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr11:	xtr10:	xtr9:	xtr8:	ytr11:	ytr10:	ytr9:	ytr8:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr7:	xtr6:	xtr5:	xtr4:	ytr7:	ytr6:	ytr5:	ytr4:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr3:	xtr2:	xtr1:	xtr0:	ytr3:	ytr2:	ytr1:	ytr0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xthr1:	xthr0:	ythr1:	ythr0:
scratch	scratch	scratch	scratch

¹ The registers are listed in order of preferred allocation by the compiler.

Table 1-27. Enhanced Communications Registers (ADSP-TS201 Processors only)¹

xtr31:	xtr30:	xtr29:	xtr28:	ytr31:	yt20:	ytr29:	ytr28:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr27:	xtr26:	xtr25:	xtr24:	ytr27:	ytr26:	ytr25:	ytr24:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr23:	xtr22:	xtr21:	xtr20:	ytr23:	ytr22:	ytr21:	ytr20:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr19:	xtr18:	xtr17:	xtr16:	ytr19:	ytr18:	ytr17:	ytr16:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr15:	xtr14:	xtr13:	xtr12:	ytr15:	ytr14:	ytr13:	ytr12:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr11:	xtr10:	xtr9:	xtr8:	ytr11:	ytr10:	ytr9:	ytr8:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr7:	xtr6:	xtr5:	xtr4:	ytr7:	ytr6:	ytr5:	ytr4:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

xtr3:	xtr2:	xtr1:	xtr0:	ytr3:	ytr2:	ytr1:	ytr0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

Table 1-27. Enhanced Communications Registers (ADSP-TS201 Processors only)¹ (Cont'd)

xthr3:	xthr2:	xthr1:	xthr0:	ythr3:	ythr2:	xthr1:	xthr0:
scratch	scratch	scratch	scratch	scratch	scratch	scratch	scratch

1 The registers are listed in order of preferred allocation by the compiler.

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 1-185](#).

This section describes:

- [“Calling Assembly Subroutines from C/C++ Programs”](#)
- [“Calling C/C++ Functions from Assembly Programs” on page 1-216](#)
- [“Using Mixed C/C++ and Assembly Naming Conventions” on page 1-217](#)
- [“C++ Programming Examples” on page 1-219](#)

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

C/C++ and Assembly Language Interface

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.

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.

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 intend to using the function from assembly programs as well, you might want your function's name to be just as you write it. Then you also need to tell the C/C++ compiler that it is 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 1-45](#)). 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 `-S` and `-O` options set, this produces the following code.



A TigerSHARC processor passes up to four arguments in registers. (The optimizer has gratuitously reversed the order of the assignments.)

```
// PROCEDURE: _asmfunc
.global _asmfunc;
_asmfunc:
    J8 = j31 + 12345;;
    [j31 + _global_p] = J6;;
    [j31 + _global_b] = XR5;;
    cjmp (NP) (ABS); [j31 + _global_a] = J4;;
```

If the arguments are on the stack, they are addressed by an offset from the stack pointer or frame pointer. For a simple function, this may be all the information you need. Do the computation, set up a return value, and return to the caller.



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.

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 1-213](#), 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

C/C++ and Assembly Language Interface

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.

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 `ccts` 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 1-28](#) shows several examples of the C/C++ and assembly interface naming conventions.

Table 1-28. 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 1-219](#)
- [“Using Complex Number Support” on page 1-220](#)

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

For example, the following command line

```
ccts -c++ source.cpp -TS101
```

runs `ccts` 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.

```
-TS101
```

Specifies that the compiler should produce code suitable for the ADSP-TS101 processor.

Using Fract Type Support

[Listing 1-1 on page 1-220](#) 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 [“C++ Fractional Type Support” on page 1-168](#).

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 ACHIEVING OPTIMAL PERFORMANCE FROM C/C++ SOURCE CODE

This chapter provides guidance in helping you to tune your application to achieve the best possible code from the compiler. Some implementation choices are available when coding an algorithm, and understanding their impact is crucial to attaining optimal performance.

This chapter contains:

- [“General Guidelines” on page 2-3](#)
- [“Loop Guidelines” on page 2-23](#)
- [“Using Built-In Functions in Code Optimization” on page 2-33](#)
- [“Smaller Applications: Optimizing for Code Size” on page 2-37](#)
- [“Pragmas” on page 2-39](#)
- [“Useful Optimization Switches” on page 2-47](#)
- [“Example: How the Optimizer Works” on page 2-48](#)
- [“Assembly Optimizer Annotations” on page 2-51](#)

The focus of what follows is on how to obtain maximal code performance from the compiler. Most of these guidelines also apply when optimizing for minimum code size, although some techniques specific to that goal are also discussed.

The first section looks at some general principles, and how the compiler can lend the most help to your optimization effort. Optimal coding styles are then considered in detail. Special features such as compiler switches, built-in functions, and pragmas are also discussed. The chapter ends with a short example to demonstrate how the optimizer works.

Small examples are included throughout this chapter to demonstrate points being made. Some show recommended coding styles, others identify styles to be avoided or code that it may be possible to improve. These are marked as “**Good**” and “**Bad**”, respectively.

General Guidelines

It is useful to bear in mind the following basic strategy when writing an application:

1. Try to choose an algorithm that is suited to the architecture being targeted. For example, there may be a trade-off between memory usage and algorithm complexity that may be influenced by the target architecture.
2. Code the algorithm in a simple, high-level generic form. Keep the target in mind, especially with respect to choices of data types.
3. You can then turn your attention towards code tuning. For critical code sections, think more carefully about the strengths of the target platform, and make any non-portable changes where necessary.

Tip: Choose the language as appropriate.

As a programmer, your first decision is to choose whether to implement your application in C or C++. This decision may be influenced by performance considerations. C++ code using only C features will have very similar performance to pure C source. Many higher-level C++ features (for example those resolved at compile-time, such as namespaces, overloaded functions and inheritance) have no performance cost. However, use of some other features may degrade performance, and so the performance loss must be weighed against the richness of expression available in C++. Examples of features that may degrade performance are virtual functions or using classes to implement basic data types.

This section contains:

- [“How the Compiler Can Help” on page 2-4](#)
- [“Data Types” on page 2-7](#)
- [“Getting the Most from IPA” on page 2-10](#)

General Guidelines

- [“Indexed Arrays vs. Pointers” on page 2-15](#)
- [“Function Inlining” on page 2-17](#)
- [“Using Inline asm Statements” on page 2-18](#)
- [“Memory Usage” on page 2-19](#)

How the Compiler Can Help

The compiler provides many facilities designed to help the programmer including compiler optimizer, PGO, and IPA optimizers.

Using the Compiler Optimizer

There is a vast difference in performance between code compiled optimized and code compiled non-optimized. In some cases optimized code can run ten or twenty times faster. Optimization should always be used when measuring performance or shipping code as product.

The optimizer in the C/C++ compiler is designed to generate efficient code from source that has been written in a straightforward manner. The basic strategy for tuning a program is to present the algorithm in a way that gives the optimizer excellent visibility of the operations and data, and hence the greatest freedom to safely manipulate the code. Future releases of the compiler will continue to enhance the optimizer, and expressing algorithms simply will provide the best chance of benefiting from such enhancements.

Note that the default setting (or “debug” mode within the VisualDSP++ IDDE) is for non-optimized compilation in order to assist programmers in diagnosing problems with their initial coding. The optimizer is enabled in VisualDSP++ by checking the **Enable optimization** checkbox under the **Project Options ->Compile** tab. This adds the `-O` (enable optimization) switch ([on page 1-39](#)) to the compiler invocation. A “release” build from within VisualDSP++ will automatically enable optimization.

Using Profile-Guided Optimization

Profile-guided optimization (PGO) is an excellent way to tune the compiler's optimization strategy for the typical run-time behavior of a program. There are many program characteristics that cannot be known statically at compile-time but can be provided by means of PGO. The compiler can use this knowledge to bring about benefits such as more accurate branch-prediction, improved loop transformations, and reduced code-size. The technique is most relevant where the behavior of the application over different data sets is expected to be very similar.

The application is first compiled with the `-pguide` switch ([on page 1-42](#)) to create an executable containing the necessary instrumentation for gathering profile data. Optimization should also be enabled; for best results, you should use the `-O` ([on page 1-39](#)) or `-ipa` ([on page 1-33](#)) switches.

The next step is to gather the profile; at present this may only be done using a simulator. To do this, run the executable with one or more training data sets. These training data sets should be representative of the data that you expect the application to process in the field. Note that unrepresentative training data sets can cause performance degradations when the application is used on real data. The profile is accumulated in a file with the extension `.pgo`. The final stage involves re-compiling the application using this gathered profile data. To do this, simply place the `.pgo` file on the command line. Optimization should also be enabled at this stage.

Note that when a `.pgo` file is placed on the command line, the `-O` optimization switch by default tries to balance between code performance and code-size considerations. More specifically, it is equivalent to using the setting `-Ov 50`. To optimize solely for performance while using PGO, the switch `-Ov 100` should be used, or the optimization slider bar (located under the **Compiler** tab in the VisualDSP++ **Project Options** dialog box) moved up to its highest setting. The `-Ov num` switch is discussed further when looking at the specific issues involved in optimization for space (see in [“Smaller Applications: Optimizing for Code Size” on page 2-37](#)).

General Guidelines

PGO should always be performed as the very last optimization step. If the application source code is changed after gathering profile data, this profile data will become invalid. The compiler will not use profile data when it can detect that it is inaccurate. However, it is possible to change source code in a way that will not be detectable to the compiler (for example, by changing constants), and it is the programmer's job to ensure that the profile data being used for optimization remains accurate.

For more details on PGO, refer to [on page 1-60](#).

Using Interprocedural Optimization

To obtain the best performance, the optimizer often requires information that can only be determined by looking outside the function that it is working on. For example, it helps to know what data can be referenced by pointer parameters, or whether a variable actually has a constant value.

The `-ipa` compiler switch ([on page 1-33](#)) enables interprocedural analysis (IPA), which can make this available. When this switch is used the compiler will be called again from the link phase to recompile the program using additional information obtained during previous compilations.

Because it only operates at link time, the effects of IPA will not be seen if you compile with the `-S` switch ([on page 1-44](#)). To see the assembly file when IPA is enabled, use the `-save-temps` switch ([on page 1-45](#)), and look at the `.s` file produced after your program has been built.

As an alternative to IPA, you can achieve many of the same benefits by adding pragma directives and other declarations such as `__builtin_aligned` to provide information to the compiler about how each function interacts with the rest of the program.

These directives are further described “[Using `\_\_builtin\_aligned` on page 2-12](#)” and “[Pragmas on page 2-39](#)”.

Data Types

The compiler directly supports the following scalar data types.

Single-Word Fixed-Point Data Types: Native Arithmetic	
char	32-bit signed integer (default) 8-bit signed integer (with -char-size-8)
unsigned char	32-bit unsigned integer (default) 8-bit unsigned integer (with -char-size-8)
short	32-bit signed integer (default) 16-bit signed integer (with -char-size-8)
unsigned short	32-bit unsigned integer (default) 16-bit unsigned integer (with -char-size-8)
int	32-bit signed integer
unsigned int	32-bit unsigned integer
long	32-bit signed integer
unsigned long	32-bit unsigned integer
long long	64-bit signed integer
unsigned long long	64-bit unsigned integer
Floating-Point Data Types: Native Arithmetic	
float	32-bit floating-point
double	32-bit floating-point (default
Floating-Point Data Types: Emulated Arithmetic	
double	64-bit floating-point (set with the -double-size-64 switch)
long double	64-bit floating-point

Fractional data types are represented using the integer types. Manipulation of these is best done by use of built-in functions, described in [“System Support Built-in Functions” on page 2-34](#).

Avoiding Emulated Arithmetic

Arithmetic operations for some types are implemented by library functions because the DSP hardware does not directly support these types. Consequently, operations for these types are far slower than native operations—sometimes by a factor of a hundred—and also produce larger code. These types are marked as “Emulated Arithmetic” in [“Data Types” on page 2-7](#).

The hardware does not provide direct support for division, so division and modulus operations are almost always multi-cycle operations, even on integral type inputs. An exception is that the compiler will convert an integral division by a power of two to a right-shift operation if the value of the divisor is known, albeit with a few fix-up instructions if the types are signed. In this case it is worth considering if an unsigned division could be used. If the compiler has to issue a full division operation, it will usually need to generate a call to a library function.

In addition, although most arithmetic operations on `long long` integers are supported by the hardware, multiplication is not. However the `long long` type is often useful for bit manipulation algorithms because of the number of bits that can be processed in each operation.

When the compiler has to generate a call to a library function for one of these arithmetic operators that are not supported by the hardware, performance will suffer not only because the operation will take multiple cycles, but also because the effectiveness of the compiler optimizer will be reduced.

For example, such an operation in a loop can prevent the compiler from making use of efficient zero-overhead hardware loop instructions. Also, calling the library to perform the required operation can change values held in scratch registers before the call, so the compiler will have to generate more stores and loads from the data stack to keep values required after the call returns. Emulated arithmetic operators should therefore be avoided where possible, especially in loops.

Using Sub-Word Types with Caution

The size of `char` and `short` integer types may be changed through use of the `-char-size-8` switch option. Although the TigerSHARC hardware does not support 8- or 16-bit loads and stores from memory, they can be simulated in code. However, it is important to remember that they are considerably less efficient than full-word memory accesses. Sub-word sized data may permit greater throughput on highly “parallelizable” inner loops, but sub-word sized scalars, or use in “non-parallelizable” loops, will degrade performance. Therefore, use sub-word sized integers with caution, or where compatibility with true byte-addressable architectures is required.

However, the TigerSHARC architecture does provide hardware support for data types such as short vectors of 16-bit and 8-bit `ints` and 16-bit fixed-point `complex`. Though not directly supported by the compiler, access to the instructions is available through intrinsic (build-in) functions.

It should be noted that these 16-bit operations are often more efficient than 32-bit ones. However, the compiler will not generate a 16-bit operation where you wrote a 32-bit operations even if it is obvious that the 16-bit operation would generate the same result. Unless using the `-char-size-8` switch ([on page 1-25](#)), the only way to get 16-bit operations is to use an build-in function. For more information on built-in functions supported by the compiler, refer to “[Compiler Built-In Functions](#)” [on page 1-98](#). Even when using `-char-size-8`, the use of intrinsics can be a more reliable way to ensure maximum throughput in critical loops.

The following code example shows scalar product written with 4x16 short vector intrinsics. It illustrates the approved style for code written with intrinsic functions.

```
typedef long long int4x16;

#define add(x,y) __builtin_add_4x16(x,y)
#define mult(x,y) __builtin_mult_i4x16(x,y)
```

General Guidelines

```
#define sum(x) __builtin_sum_4x16(x)
int sp4x16(int4x16 a[], int4x16 b[], int n)
{
    int i;
    int4x16 sum4 = 0;
    for (i = 0; i < n/4; ++i)
        sum4 = add(sum4, mult(a[i], b[i]));
    return sum(sum4);
}
```

Good: uses 4x16 short vector intrinsics.

Getting the Most from IPA

Interprocedural analysis (IPA) is designed to try to propagate information about the program to parts of the optimizer that can use it. This section looks at what information is useful, and how to structure your code to make this information easily accessible to the analysis.

Initialize Constants Staticly

IPA will identify variables that have only one value and replace them with constants, resulting in a host of benefits for the optimizer's analysis. For this to happen a variable must have a single value throughout the program. If the variable is statically initialized to zero, as all global variables are by default, and is subsequently assigned some other value at another point in the program, then the analysis sees two values and will not consider the variable to have a constant value. For example,

```
#include <stdio.h>
int val;           // initialized to zero

void init() {
    val = 3;        // re-assigned
}

void func() {
```

Achieving Optimal Performance from C/C++ Source Code

```
    printf("val %d",val);  
}  
  
int main() {  
    init();  
    func();  
}
```

Bad: IPA cannot see that `val` is a constant

is better written as

```
#include <stdio.h>  
const int val = 3;    // initialized once  
  
void init() {  
}  
  
void func() {  
    printf("val %d",val);  
}  
  
int main() {  
    init();  
    func();  
}
```

Good: IPA knows `val` is 3.

Quad-Word-Aligning Your Data

To make most efficient use of the hardware, it must be kept fed with data. In many algorithms, the balance of data accesses to computations is such that, to keep the hardware fully utilized, data must be fetched with 32-bit loads.

The hardware requires that references to memory be naturally aligned. Therefore, 64-bit references must be at even address locations, and 128-bit references at quad-word-aligned addresses. For the most efficient code to be generated, you should ensure that data buffers are quad-word-aligned.

General Guidelines

The compiler helps to establish the alignment of array data. The stack frames are kept quad-word-aligned. Top-level arrays are allocated at quad-word-aligned addresses, regardless of their data types. However, arrays within structures are not aligned beyond the required alignment for their type. It may be worth using the `#pragma align n` directive to force the alignment of arrays in this case.

If you write programs that only pass the address of the first element of an array as a parameter and loops that process these input arrays an element at a time, starting at element zero, then IPA should be able to establish that the alignment is suitable for full-width accesses.

Where an inner loop processes a single row of a multi-dimensional array, try to ensure that each row begins on a word boundary. This may require the insertion of dummy data at the end of each row to make the row length a multiple of four words.

Using `__builtin_aligned`

To avoid the need to use the `-ipa` option to propagate alignment, and for situations when IPA cannot guarantee the alignment but you can, it is possible to use the `__builtin_aligned` statement to assert the alignment of important pointers, meaning that the pointer points to data that is aligned.

The assertion is particularly useful for function parameters, although you may assert that any pointer is aligned. For example, when compiling the function:

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

Bad: without IPA, compiler doesn't know the alignment of `a` and `b`.

the compiler does not know the alignment of pointers `a` and `b` if IPA is not being used. However, by modifying the function to:

```
void copy(char *a, char *b) {
    int i;
    __builtin_aligned(a, 4);
    __builtin_aligned(b, 4);
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```

Good: both pointer parameters are known to be aligned.

the compiler can be told that the pointers are aligned on quad-word boundaries. To assert instead that both pointers are always aligned one char past a quad-word boundary, use:

```
void copy(char *a, char *b) {
    int i;
    __builtin_aligned(a+1, 4);
    __builtin_aligned(b+1, 4);
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```

Good: both pointer parameters are known to be misaligned.

The expression used as the first parameter to the built-in function obeys the usual C rules for pointer arithmetic. However, the second parameter to the built-in function should give the alignment in words. The exception is that when using the `-char-size-8` option, the alignment should instead be given in bytes. In that case, to specify that a pointer `a` is quad-word aligned, use `__builtin_aligned(a, 16)`.

When a loop has up to two non-aligned pointers, the compiler can make use of the hardware data alignment buffers to access 128-bits of data from those pointers. Stores to memory via misaligned pointers are more difficult, so it is more important to store to an aligned buffer than to load from one.

Avoiding Aliases

It may seem that the iterations may be performed in any order in the following loop:

```
void fn(char a[], char b[], int n) {  
    for (i = 0; i < n; ++i)  
        a[i] = b[i];  
}
```

Bad: *a* and *b* may alias each other.

but *a* and *b* are both parameters, and, although they are declared with [], they are in fact pointers, which may point to the same array. When the same data may be reachable through two pointers, they are said to alias each other.

If the `-ipa` switch is enabled, the compiler will look at the call sites of *fn* and try to determine whether *a* and *b* can ever point to the same array.

Even with IPA, it is quite easy to create what appear to the compiler as aliases. The analysis works by associating pointers with sets of variables that they may refer to at some point in the program. If the sets for two pointers are found to intersect, then both pointers are assumed to point to the union of the two sets.

If *fn* above were called in two places with global arrays as arguments, then IPA would have the results shown below:

```
fn(glob1, glob2, N);  
fn(glob1, glob2, N);
```

Good: sets for *a* and *b* do not intersect: *a* and *b* are not aliases.

```
fn(glob1, glob2, N);  
fn(glob3, glob4, N);
```

Good: sets for *a* and *b* do not intersect: *a* and *b* are not aliases.

Achieving Optimal Performance from C/C++ Source Code

```
fn(glob1, glob2, N);  
fn(glob3, glob1, N);
```

Bad: sets intersect; *a* and *b* may be aliases.

The third case arises because IPA considers the union of all calls at once, rather than considering each call individually, when determining whether there is a risk of aliasing. If each call were considered individually, IPA would have to take flow control into account and the number of permutations would make compilation time impracticably long.

The lack of control flow analysis can also create problems when a single pointer is used in multiple contexts. For example, it is better to write:

```
int *p = a;  
int *q = b;  
    // some use of p  
    // some use of q
```

Good: *p* and *q* do not alias.

than

```
int *p = a;  
    // some use of p  
p = b;  
    // some use of p
```

Bad: uses of *p* in different contexts may alias.

because the latter may cause extra apparent aliases between the two uses.

Indexed Arrays vs. Pointers

C allows a program to access data from an array in two ways: either by indexing from an invariant base pointer, or by incrementing a pointer. These two versions of vector addition illustrate the two styles:

General Guidelines

Style 1: using indexed arrays

```
void va_ind(const short a[], const short b[], short out[], int n) {
    int i;
    for (i = 0; i < n; ++i)
        out[i] = a[i] + b[i];
}
```

Style 2: using pointers

```
void va_ptr(const short a[], const short b[], short out[], int n) {
    int i;
    short *pout = out;
    const short *pa = a, *pb = b;
    for (i = 0; i < n; ++i)
        *pout++ = *pa++ + *pb++;
}
```

Trying Pointer and Indexed Styles

One might hope that the chosen style would not make any difference to the generated code, but this is not always the case. Sometimes, one version of an algorithm will generate better optimized code than the other, but it is not always the same style that is better.

Tip: Try both pointer and index styles.

The pointer style introduces additional variables that compete with the surrounding code for resources during the compiler optimizer's analysis. Array accesses, on the other hand, must be transformed to pointers by the compiler and sometimes it does not do the job as well as you could do by hand.

The best strategy is to start with array notation. If the generated code looks unsatisfactory, try using pointers. Outside the critical loops, use the indexed style, since it is easier to understand.

Function Inlining

The function inlining may be used in two ways

- By annotating functions in the source code with the `inline` keyword. In this case, function inlining is only performed when optimization is enabled.
- By turning on automatic inlining with the `-Oa` switch ([on page 1-39](#)). This switch automatically enables optimization.

Tip: Inline small, frequently executed functions.

You can use the compiler's `inline` keyword to indicate that functions should have code generated inline at the point of call. Doing this avoids various costs such as program flow latencies, function entry and exit instructions and parameter passing overheads. Using an inline function also has the advantage that the compiler can optimize through the inline code and does not have to assume that scratch registers and condition states are modified by the call. Prime candidates for inlining are small, frequently used functions because they will cause the least code-size increase while giving most performance benefit.

As an example of the usage of the `inline` keyword, the function below sums two input parameters and returns the result.

```
inline int add(int a, int b) {  
    return (a+b);  
}
```

Good: use of the `inline` keyword.

Inlining has a code-size to performance trade-off that should be considered when it is used. With `-Oa`, the compiler will automatically inline small functions where possible. If the application has a tight upper code-size limit, the resulting code-size expansion may be too great. It is worth considering using automatic inlining in conjunction with the `-Ov num` switch ([on page 1-40](#)) to restrict inlining (and other optimiza-

tions with a code-size cost) to parts of the application that are performance-critical. This will be considered in more detail later in this chapter.

Using Inline `asm` Statements

The compiler allows use of inline `asm` statements to insert small sections of assembly code into C code.



Avoid use of inline `asm` statements where built-in functions may be used instead

The compiler does not intensively optimize code that contains inline `asm` statements because it has little understanding about what the code in the statement does. In particular, use of an `asm` statement in a loop may inhibit useful transformations.

The compiler has been enhanced with a large number of built-in functions. These generate specific hardware instructions and are designed to allow the programmer to more finely tune the code produced by the compiler, or to allow access to system support functions. A complete list of compiler's built-in functions is given in [“Compiler Built-In Functions” on page 1-98](#).

Use of these built-in functions is much preferred to using inline `asm` statements. Since the compiler knows what each built-in function does, it can easily optimize around them. Conversely, since the compiler does not parse `asm` statements, it does not know what they do, and so is hindered in optimizing code that uses them. Note also that errors in the text string of an `asm` statement will be caught by the assembler and not the compiler.

Examples of efficient use of built-in functions are given in [“System Support Built-in Functions” on page 2-34](#).

Memory Usage

The compiler, in conjunction with the use of the linker description file (.LDF), allows the programmer control over where data is placed in memory. This section describes how to best lay out data for maximum performance.

Putting Arrays into Different Memory Sections

The DSP hardware can support two memory operations on a single instruction line, combined with a compute instruction. However, two memory operations will only complete in one cycle if the two addresses are situated in different memory blocks; if both access the same block, then a stall will be incurred.

Tip: Try to put arrays into different memory sections.

Take as an example the dot product loop below. Because data is loaded from both array *a* and array *b* in every iteration of the loop, it may be useful to ensure that these arrays are located in different blocks.

```
for (i=0; i<100; i++)  
    sum += a[i] * b[i];
```

Bad: compiler assumes that two memory accesses together may give a stall.

The efficient memory usage is facilitated using the “Dual Memory Support Language Keywords” compiler extension (see “[Memory Support Keywords \(pm dm\)](#)” on page 1-88). Placing a *pm* qualifier before the type definition tells the compiler that the array is located in what is notionally called “Program Memory” (*pm*).

The memory of a TigerSHARC processor is one unified address space and there is no restriction concerning in which part of memory program code or data can be placed into. However, the default .LDF files ensure that

General Guidelines

pm-qualified data is placed in a different memory block than non-qualified (or dm-qualified) data, thus allowing two accesses to occur simultaneously without incurring a stall.

The array declaration of one of either a or b is modified to

```
pm int a[100];
```

and any pointers to the buffer a become, for example,

```
pm int *p = a;
```

It is also possible to use the .LDF file to place data in different user-defined memory sections. This has the advantage that it is possible to define more than two sections which do not conflict with each other. Consider again the example above.

First, let's define two memory banks in the MEMORY portion of the .LDF file, such as in the following example.

Example: MEMORY portion of LDF modified to define memory banks.

```
MEMORY {  
    BANK_A1 {  
        TYPE(RAM) WIDTH(32)  
        START(start_address_1) END(end_address_1)  
    }  
    BANK_A2 {  
        TYPE(RAM) WIDTH(32)  
        START(start_address_2) END(end_address_2)  
    }  
}
```

Then, configure the SECTIONS portion to tell the linker to place data sections in specific memory banks.

Achieving Optimal Performance from C/C++ Source Code

Example: SECTIONS portion of LDF modified to define memory banks.

```
SECTIONS {
    bank_a1 {
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS($OBJECTS(bank_a1))
    } >BANK_A1
    bank_a2 {
        INPUT_SECTION_ALIGN(4)
        INPUT_SECTIONS($OBJECTS(bank_a2)) } >BANK_A2
}
```

In the C source code, declare arrays with the `section("section_name")` construct preceding a buffer declaration; in this case

```
section("bank_a1") short a[100];
section("bank_a2") short b[100];
```

This ensures that the two array accesses may occur simultaneously without incurring a stall.

Any pointers to the buffers `a` and `b` should likewise be modified:

```
section("bank_a1") short *p = a;
```

Note that the explicit placement of data in sections or in Program Memory can only be done for global data.

Using the Bank Qualifier

The bank qualifier can also be useful to write functions which make use of the fact that buffers are placed in separate memory blocks. For example, it might be useful to create a function

```
void func(int bank("red") *p, int bank("blue") *q) {
    // some code
}
```

Good: uses bank qualifier to allow simultaneous access to `p` and `q`.

General Guidelines

if you would like to call `func` in different places, but always with pointers to buffers in different sections of memory. The bank qualifier symbolizes that the buffers are in different sections without requiring that the sections themselves be specified.

Therefore, `func` (in the example above) may be called with the first parameter pointing to memory in `section("bank_a1")` and the second pointing to data in `section("bank_a2")` or vice versa.

Loop Guidelines

Loops are where an application will ordinarily spend the majority of its time. It is therefore useful to look in detail at how to help the compiler to produce the most efficient code possible for them.

Keeping Loops Short

For best code efficiency, loops should be as small as possible. Large loop bodies are usually more complex and difficult to optimize. Additionally, they may require register data to be stored in memory. This will cause a decrease in code density and execution performance.

Avoiding Unrolling Loops

Tip: Do not unroll loops yourself.

Not only does loop unrolling make the program harder to read but it also prevents optimization by complicating the code for the compiler.

```
void va1(const short a[], const short b[], short c[], int n)
{
    int i;
    for (i = 0; i < n; ++i) {
        c[i] = b[i] + a[i];
    }
}
```

Good: the compiler will unroll if it helps.

```
void va2(const short a[], const short b[], short c[], int n)
{
    short xa, xb, xc, ya, yb, yc;
    int i;
    for (i = 0; i < n; i+=2) {
        xb = b[i]; yb = b[i+1];
        xa = a[i]; ya = a[i+1];
    }
}
```

Loop Guidelines

```
        xc = xa + xb; yc = ya + yb;  
        c[i] = xc; c[i+1] = yc;  
    }  
}
```

Bad: harder for the compiler to optimize.

Avoiding Loop-Carried Dependencies

A loop-carried dependency exists when computations in a given iteration of a loop cannot be completed without knowledge of the values calculated in earlier iterations. When a loop has such dependencies, the compiler cannot overlap loop iterations. Some dependencies are caused by scalar variables that are used before they are defined in a single iteration. However, an optimizer can reorder iterations in the presence of the class of scalar dependencies known as *reductions*. These are loops that reduce a vector of values to a scalar value using an associative and commutative operator—a common case being multiply and accumulate.

```
for (i = 0; i < n; ++i)  
    x = a[i] - x;
```

Bad: loop-carried dependence is a scalar dependency.

```
for (i = 0; i < n; ++i)  
    x += a[i] * b[i];
```

Good: loop-carried dependence is a reduction.

In the first case, the scalar dependency is the subtraction operation; the variable `x` is modified in a manner that gives different results if the iterations are performed out of order. In contrast, in the second case the properties of the addition operator stipulate that the compiler can perform the iterations in any order and still get the same result. Other examples of operators which are reductions are `bitwise and/or`, and `min/max`. In par-

ticular, the existence of loop-carried dependencies that are not reductions is one criterion that prevents the compiler from being able to vectorize a loop—that is, to execute more than one iteration concurrently.

Floating-point addition is by default assumed to obey the criteria necessary to be considered a reduction. However, strictly speaking, rounding effects can change the result when the order of summation is varied.

Behavior consistent with the original program can be regained using the `-no-fp-associative` compiler switch (see [on page 1-37](#)).

Avoiding Loop Rotation by Hand

Tip: Do not rotate loops by hand.

Programmers are often tempted to “rotate” loops in DSP code by “hand” attempting to execute loads and stores from earlier or future iterations at the same time as computation from the current iteration. This technique introduces loop-carried dependencies that prevent the compiler from rearranging the code effectively. However, it is better to give the compiler a “normalized” version, and leave the rotation to the compiler.

```
int ss(short *a, short *b, int n) {
    int sum = 0;
    int i;
    for (i = 0; i < n; i++) {
        sum += a[i] + b[i];
    }
    return sum;
}
```

Good: will be rotated by the compiler.

```
int ss(short *a, short *b, int n) {
    short ta, tb;
    int sum = 0;
    int i = 0;
    ta = a[i]; tb = b[i];
    for (i = 1; i < n; i++) {
```

Loop Guidelines

```
        sum += ta + tb;  
        ta = a[i]; tb = b[i];  
    }  
    sum += ta + tb;  
    return sum;  
}
```

Bad: rotated by hand—hard for the compiler to optimize.

By rotating the loop, the scalar variables `ta` and `tb` have been added, introducing loop-carried dependencies.

Avoiding Array Writes in Loops

Other dependencies can be caused by writes to array elements. In the following loop, the optimizer cannot determine whether the load from `a` reads a value defined on a previous iteration or one that will be overwritten in a subsequent iteration.

```
for (i = 0; i < n; ++i)  
    a[i] = b[i] * a[c[i]];
```

Bad: has array dependency.

The optimizer can resolve access patterns where the addresses are expressions that vary by a fixed amount on each iteration. These are known as “induction variables”.

```
for (i = 0; i < n; ++i)  
    a[i+4] = b[i] * a[i];
```

Good: uses induction variables.

Inner Loops vs. Outer Loops

Tip: Inner loops should iterate more than outer loops.

The optimizer focuses on improving the performance of inner loops because this is where most programs spend the majority of their time. It is considered a good trade-off for an optimization to slow down the code before and after a loop if it is going to make the loop body run faster.

Therefore, try to make sure that your algorithm also spends most of its time in the inner loop; otherwise it may actually be made to run slower by optimization. If you have nested loops where the outer loop runs many times and the inner loop a small number of times, it may be possible to rewrite the loops so that the outer loop has the fewer iterations.

Avoiding Conditional Code in Loops

If a loop contains conditional code, control-flow latencies may incur large penalties if the compiler has to generate conditional jumps within the loop. In some cases, the compiler will be able to convert `IF-THEN-ELSE` and `?` constructs into conditional instructions. In other cases, it will be able to relocate the expression evaluation outside of the loop entirely. However, for important loops, linear code should be written where possible.

There are several tricks that can be used to try to remove the necessity for conditional code. For example, there is hardware support for `min` and `max`. The compiler will usually succeed in transforming conditional code equivalent to `min` or `max` into the single instruction. With particularly convoluted code the transformation may be missed, in which case it is better to use `min` or `max` in the source code.

The compiler will not perform the loop transformation that interchanges conditional code and loop structures. Instead of writing

```
for (i=0; i<100; i++) {  
    if (mult_by_b)
```

Loop Guidelines

```
        sum1 += a[i] * b[i];
    else
        sum1 += a[i] * c[i];
}
```

Bad: loop contains conditional code.

it is better to write

```
if (mult_by_b) {
    for (i=0; i<100; i++)
        sum1 += a[i] * b[i];
} else {
    for (i=0; i<100; i++)
        sum1 += a[i] * c[i];
}
```

Good: two simple loops can be optimized well.

if this is an important loop.

Avoiding Placing Function Calls in Loops

The compiler will not usually be able to generate a hardware loop if the loop contains a function call due to the expense of saving and restoring the context of a hardware loop. In addition to obvious function calls, such as `printf()`, hardware loop generation can also be prevented by operations such as division, modulus, and some type coercions.

These operations may require implicit calls to library functions. For more details, see [“Data Types” on page 2-7](#).

Avoiding Non-Unit Strides

If you write a loop, such as

```
for (i=0; i<n; i+=3) {  
    // some code  
}
```

Bad: non-unit stride means division may be required.

then in order for the compiler to turn this into a hardware loop, it will need to work out the loop trip count. To do so, it must divide n by 3. The compiler will decide that this is worthwhile as it will speed up the loop, but as discussed above, division is an expensive operation. Try to avoid creating loop control variables with strides of non-unit magnitude.

In addition, try to keep memory accesses in consecutive iterations of an inner loop contiguous. This is particularly applicable to multi-dimensional arrays. Therefore,

```
for (i=0; i<100; i++)  
    for (j=0; j<100; j++)  
        sum += a[i][j];
```

Good: memory accesses contiguous in inner loop.

is likely to be better than

```
for (i=0; i<100; i++)  
    for (j=0; j<100; j++)  
        sum += a[j][i];
```

Bad: loop cannot be unrolled to use wide loads.

as the former is more amenable to vectorization.

Similarly, two-dimensional arrays should be defined in a single block of memory rather than as an array of pointers to rows, all separately allocated with `malloc`. It is difficult for the compiler to keep track of the alignment of the pointers of the latter.

Loop Control

Tip: Use `int` types for loop control variables and array indices.

Tip: Use automatic variables for loop control and loop exit test.

For loop control variables and array indices, it is always better to use signed `ints` rather than any other integral type. The C standard requires various type promotions and standard conversions that complicate the code for the compiler optimizer. Frequently, the compiler is still able to deal with such code and create hardware loops and pointer induction variables. However, it does make it more difficult for the compiler to optimize and may occasionally result in under-optimized code.

The same advice goes for using automatic (local) variables for loop control. It is easy for a compiler to see that an automatic scalar, whose address is not taken, may be held in a register during a loop. But it is not as easy when the variable is a global or a function static. Therefore, code such as

```
for (i=0; i<globvar; i++)  
    a[i] = 10;
```

Bad: may need to reload `globvar` on every iteration.

may not create a hardware loop if the compiler cannot be sure that the write into the array `a` does not change the value of the global variable. The `globvar` must be re-loaded each time around the loop before performing the exit test.

In this circumstance, the programmer can make the compiler's job easier by writing:

```
int upper_bound = globvar;  
for (i=0; i<upper_bound; i++)  
    a[i] = 10;
```

Good: easily becomes hardware loop.

Using the Restrict Qualifier

The `restrict` qualifier provides one way to help the compiler resolve pointer aliasing ambiguities. Accesses from distinct `restricted` pointers do not interfere with each other. The loads and stores in the following loop

```
for (i=0; i<100; i++)  
    a[i] = b[i];
```

Bad: possible alias of arrays `a` and `b`.

may be disambiguated by writing

```
int * restrict p = a;  
int * restrict q = b;  
for (i=0; i<100; i++)  
    *p++ = *q++;
```

Good: `restrict` qualifier tells compiler that memory accesses do not alias.

The `restrict` keyword is particularly useful on function parameters.

Using the Const Qualifier

By default, the compiler assumes that the data referenced by a pointer to `const` type will not change. Therefore, another way to tell the compiler that the two arrays `a` and `b` do not overlap is to use the `const` keyword.

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

Good: pointers disambiguated via `const` qualifier.

Loop Guidelines

The use of `const` in the above example will have a similar effect on the `no_alias` pragma (see in “[#pragma no_alias](#)” on page 2-46). In fact, the `const` implementation is better since it also allows the optimizer to use the fact that accesses via `a` and `b` are independent in other parts of the code, not just the inner loop.

In C, it is legal, though bad programming practice, to use casts to allow the data pointed to by pointers to `const` type to change. This should be avoided since, by default, the compiler will generate code that assumes `const` data does not change. If you have a program that modifies `const` data through a pointer, you can generate standard-conforming code by using the compile-time flag `-const-read-write` (see [on page 1-25](#)).

Avoiding Long Latencies

All pipelined machines will introduce stall cycles when you cannot execute the current instruction until a prior instruction has exited the pipeline. For example, the TigerSHARC processor stalls for four cycles on a table lookup; `a[b[i]]` takes four cycles more than you would expect.

If a stall is seen empirically, but it is not obvious to you exactly why it is occurring, a good way to learn about the cause is the **Pipeline Viewer**. This can be accessed through **Debug Windows -> Pipeline Viewer** in the VisualDSP++ 3.5 IDDE. By single-stepping through the program, you will see where the stall occurs. Note that the **Pipeline Viewer** is only available within a simulator session.

Using Built-In Functions in Code Optimization

Built-in functions, also known as compiler intrinsics, provide a method for the programmer to efficiently use low-level features of the DSP hardware while programming in C. Although this section does not cover all the built-in functions available (for more information, refer to [“Compiler Built-In Functions” on page 1-98](#)), it presents some code examples where implementation choices are available to the programmer.

Using Fractional Data

Fractional data, represented as an integral type, can be manipulated in two ways. To give you the most control over your data, use the intrinsics. Let us consider the fractional dot product algorithm. This may be written as:

```
short dot_product (short *a, short *b) {
    int i;
    short sum=0;
    for (i=0; i<100; i++) {
        /* this line is performance critical */
        sum += ((a[i]*b[i]) >> 15);
    }
    return sum;
}
```

Bad: uses shifts to implement fraction multiplication.

However, this presents some problems to the optimizer. Normally, the code generated here would be a multiply, followed by a shift, followed by an accumulation. However, the DSP hardware has a fractional multiply accumulate instruction that performs all these tasks in one cycle.

Using Built-In Functions in Code Optimization

The recommended coding style is to use compiler intrinsics (built-in functions). In the following example, an intrinsic is used to multiply fractional 16-bit data. Note that it is also assumed that it has been possible to rearrange the 16-bit fractional data to be packed two items per word.

```
#include <math.h>
short dot_product(int4x16 *a, int4x16 *b) {
    int i;
    int4x16 sum=0;
    for (i=0; i<100; i++) {
        /* this line is performance critical */
        sum = __builtin_add_4x16(
            sum, __builtin_mult_fr4x16(a[i],b[i]));
    }
    return __builtin_sum_4x16(sum);
}
```

Good: uses intrinsics to implement fractional multiplication.

Note that the `int4x16` type, along with other related ones, are merely typedefs to C integer types used by convention in standard include files. The compiler does not have any in-built knowledge of these types and treats them exactly as the integer types that they are typedefed to.

System Support Built-in Functions

Built-in functions allow to perform low-level system management, in particular for the manipulation of system registers (defined in `sysreg.h`). It is usually better to use these built-in functions rather than inline `asm` statements.

The built-in functions cause the compiler to generate efficient inline instructions and their use often results in better optimization of the surrounding code at the point where they are used. Using built-in functions will also usually result in improved code-readability. For more information on built-in functions supported by the compiler, refer to [“Compiler Built-In Functions” on page 1-98](#).

Using Circular Buffers

Circular buffers are often extremely useful in DSP code. They can be used in several ways. Consider the C code:

```
for (i=0; i<1000; i++) {  
    sum += a[i] * b[i%20];  
}
```

Good: the compiler knows that `b` is accessed as a circular buffer.

Clearly the access to array `b` is a circular buffer. When optimization is enabled, the compiler will produce a hardware circular buffer instruction for this access.

However, consider the slightly more complicated example:

```
for (i=0; i<1000; i+=n) {  
    sum += a[i] * b[i%20];  
}
```

Bad: may not be able to use circular buffer to access `b`.

In this case, the compiler does not know if `n` is positive and less than 20. If it is, then the access may be correctly implemented as a hardware circular buffer. On the other hand, if it is greater than 20, a circular buffer increment may not yield the same results as the C code.

The programmer has two options here. One is to compile with the `-force-circbuf` switch (on [page 1-30](#)). This tells the compiler that any access of the form `a[i%n]` should be considered as a circular buffer. Before using this switch, you should check that this assumption is valid for your application.

The preferred option, however, is to use built-in functions to perform the circular buffering. Two functions (`__builtin_circindex` and `__builtin_circptr`) are provided for this purpose.

Using Built-In Functions in Code Optimization

To make it clear to the compiler that a circular buffer should be used, you may write either:

```
for (i=0, j=0; i<1000; i+=n) {  
    sum += a[i] * b[j];  
    j = __builtin_circindex(j, n, 20);  
}
```

Good: explicit use of circular buffer via `__builtin_circindex`.

or

```
int *p = b;  
for (i=0, j=0; i<1000; i+=n) {  
    sum += a[i] * (*p);  
    p = __builtin_circptr(p, n, b, 80);  
}
```

Good: explicit use of circular buffer via `__builtin_circptr`.

For more information, refer to [“Compiler Built-In Functions” on page 1-98](#)).

Smaller Applications: Optimizing for Code Size

The same ethos for producing fast code also applies to producing small code. You should present the algorithm in a way that gives the optimizer excellent visibility of the operations and data, and hence the greatest freedom to safely manipulate the code to produce small applications.

Once the program is presented in this way, the optimization strategy will depend on the code-size constraint that the program must obey. The first step should be to optimize the application for full performance, using `-O` or `-ipa` switches. If this obeys the code-size constraints, then no more need be done.

The “optimize for space” switch `-Os` ([on page 1-39](#)), which may be used in conjunction with IPA, will perform every performance-enhancing transformation except those that increase code-size. In addition, the `-e` linker switch (`-flags-link -e` if used from the compiler command line) may be helpful ([on page 1-29](#)). This performs section elimination in the linker to remove unneeded data and code. If the code produced with `-Os` and `-e` does not meet the code-size constraint, some analysis of the source code will be required to try to reduce the code size further.

Note that loop transformations such as unrolling and software pipelining increase code size. But it is these loop transformations that also give the greatest performance benefit. Therefore, in many cases compiling for minimum code size will produce significantly slower code than optimizing for speed.

The compiler provides a way to balance between the two extremes of `-O` and `-Os`. This is the sliding-scale `-Ov num` switch (adjustable using the optimization slider bar under **Project Options** in the VisualDSP++ IDE), described [on page 1-9](#). The `num` parameter is a value between 0 and 100, where the lower value corresponds to minimum code size and the upper to maximum performance. A value in between will try to opti-

mize the frequently executed regions of code for maximum performance, while keeping the infrequently executed parts as small as possible. The switch is most reliable when using profile-guided optimization (see “[Optimization Control](#)” on page 1-60) since the execution counts of the various code regions have been measured experimentally. Without PGO, the execution counts are estimated, based on the depth of loop nesting.



Avoid the use of inline code.

Avoid using the `inline` keyword to inline code for functions that are used a number of times, especially if they not very small functions. The `-Os` switch does not have any effect on the use of the `inline` keyword. It does, however, prevent automatic inlining (using the `-Oa` switch) from increasing the code size. Macro functions can also cause code expansion and should be used with care.

Pragmas

Pragmas can assist optimization by allowing the programmer to make assertions or suggestions to the compiler. This section looks at how they can be used to finely tune source code. Refer to [“Pragmas” on page 1-148](#) for full details of how each pragma works; the emphasis here will be in considering under what circumstances they are useful during the optimization process.

In most cases the pragmas serve to give the compiler information which it is unable to deduce for itself. It must be emphasized that the programmer is responsible for making sure that the information given by the pragma is valid in the context in which it is used. Use of a pragma to assert that a function or loop has a quality that it does not in fact have is likely to result in incorrect code and hence a malfunctioning application.

An advantage of the use of pragmas is that they allow code to remain portable, since they will normally be ignored by a compiler that does not recognize them.

Function Pragmas

Function pragmas include `#pragma const`, `#pragma pure`, `#pragma regs_clobbered`, and `#pragma optimize_{off|for_speed|for_space|as_cmd_line}`.

#pragma const

The `pragma const` pragma asserts to the compiler that a function does not have any side effects (such as modifying global variables or data buffers), and the result returned is only a function of the parameter values. The pragma may be applied to a function prototype or definition. It helps the compiler since two calls to the function with identical parameters will always yield the same result. In this way, calls to `#pragma const` functions may be hoisted out of loops if their parameters are loop independent.

Pragmas

#pragma pure

Like `#pragma const`, the `#pragma pure` asserts to the compiler that a function does not have any side effects (such as modifying global variables or data buffers). However, the result returned may be a function of both the parameter values and any global variables. The pragma may be applied to a function prototype or definition. Two calls to the function with identical parameters will always yield the same result provided that no global variables have been modified between the calls. Hence, calls to `#pragma pure` functions may be hoisted out of loops if their parameters are loop independent and no global variables are modified in the loop.

#pragma regs_clobbered

The `regs_clobbered` pragma is a useful way to improve the performance of code that makes function calls. The best use of the pragma is to increase the number of call-preserved registers available across a function call. There are two complementary ways in which this may be done.

First of all, suppose that you have a function written in assembly that you wish to call from C source code. The `regs_clobbered` pragma may be applied to the function prototype to specify which registers are “clobbered” by the assembly function, that is, which registers may have different values before and after the function call. Consider for example an simple assembly function to add two integers and mask the result to fit into 8 bits:

```
_add_mask:
    J8 = J4 + J5;;
    J0 = 255;;
    J8 = J8 AND J0;;
    cjmp (ABS);;
._add_mask.end:
```

Achieving Optimal Performance from C/C++ Source Code

Clearly the function does not modify the majority of the scratch registers available and thus these could instead be used as call-preserved registers. In this way, fewer spills to the stack would be needed in the caller function. Using the prototype

```
#pragma regs_clobbered "J0, J8, jICC"
int add_mask(int, int);
```

Good: uses `regs_clobbered` to increase call-preserved register set.

the compiler is told which registers are modified by a call to the `add_mask` function. The registers not specified by the pragma are assumed to preserve their values across such a call and the compiler may use these spare registers to its advantage when optimizing the call sites.

The pragma is also powerful when all of the source code is written in C. In the above example, a C implementation might be:

```
int add_mask(int a, int b) {
    return ((a+b)&255);
}
```

Bad: function thought to clobber entire volatile register set.

Since this function will not need many registers when compiled, it can be defined using

```
#pragma regs_clobbered "J0, J8, CCset"
int add_mask(int a, int b) {
    return ((a+b)&255);
}
```

Good: function compiled to preserve most registers.

to ensure that any other registers aside from `J0`, `J8` and the condition codes will not be modified by the function. If any other registers are used in the compilation of the function, they will be saved and restored during the function prologue and epilogue.

Pragmas

In general, it is not very helpful to specify any of the condition codes as call-preserved as they are difficult to save and restore and are usually clobbered by any function. Moreover, it is usually of limited benefit to be able to keep them live across a function call. Therefore, it is better to use `CCset` (all condition codes) rather than `jICC` in the clobbered set above. For more information, refer to “[#pragma regs_clobbered string](#)” on page 1-158.

#pragma optimize_{off | for_speed | for_space}

The `optimize_` pragma may be used to change the optimization setting on a function-by-function basis. In particular, it may be useful to optimize functions that are rarely called (for example, error handling code) for space (using `#pragma optimize_for_space`), whereas functions critical to performance should be compiled for maximum speed (using `#pragma optimize_for_speed`). The `#pragma optimize_off` is useful for debugging specific functions without increasing the size or decreasing the performance of the overall application unnecessarily.

The `#pragma optimize_as_cmd_line` resets the optimization settings to be those specified on the `ccts` command line when the compiler was invoked. Refer to “[General Optimization Pragmas](#)” on page 1-162 for more information on pragmas.

Loop Optimization Pragmas

Many pragmas are targeted towards helping to produce optimal code for inner loops. These are the `loop_count`, `no_vectorization`, `vector_for`, `all_aligned`, `different_banks`, and `no_alias` pragmas.

#pragma loop_count

The `loop_count` pragma enables the programmer to inform the compiler about a loop’s iteration count. The compiler is able to make more reliable decisions about the optimization strategy for a loop if it knows the iteration count range. If you know that the loop count is always a multiple of

some constant, this can also be useful as it allows a loop to be partially unrolled or vectorized without the need for conditionally-executed iterations.

Knowledge of the minimum trip count may allow the compiler to omit the guards that are usually required after software pipelining. Any of the parameters of the pragma that are unknown may be left blank.

An example of the use of the `loop_count` pragma might be:

```
#pragma loop_count( /*minimum*/ 40, /*maximum*/ 100, /*modulo*/ 4)
for (i=0; i<n; i++)
    a[i] = b[i];
```

Good: the `loop_count` pragma gives compiler helpful information to assist optimization.

For more information, refer to “[#pragma loop_count\(min, max, modulo\)](#)” on page 1-154.

#pragma no_vectorization

Vectorization (executing more than one iteration of a loop in parallel) can slow down loops with very small iteration counts since a loop prologue and epilogue are required. The `no_vectorization` pragma can be used directly above a `for` or `do` loop to tell the compiler *not* to vectorize the loop.

#pragma vector_for

The `vector_for` pragma is used to help the compiler to resolve dependencies that would normally prevent it from vectorizing a loop. It tells the compiler that all iterations of the loop may be run in parallel with each other, subject to rearrangement of reduction expressions in the loop. In other words, there are no loop-carried dependencies except reductions. An

Pragmas

optional parameter, *n*, may be given in parentheses to say that only *n* iterations of the loop may be run in parallel. The parameter must be a literal value. For example,

```
for (i=0; i<100; i++)
    a[i] = b[i] + a[i-4];
```

Bad: cannot be vectorized due to possible alias between *a* and *b*.

cannot be vectorized if the compiler cannot tell that the array *b* does not alias array *a*. But the pragma may be added to tell the compiler that in this case four iterations may be executed concurrently.

```
#pragma vector_for (4)
for (i=0; i<100; i++)
    a[i] = b[i] + a[i-4];
```

Good: `pragma vector_for` disambiguates alias.

Note that this pragma does not force the compiler to vectorize the loop; the optimizer will check various properties of the loop and will not vectorize it if it believes that it is unsafe or cannot deduce information necessary to carry out the vectorization transformation. The pragma assures the compiler that there are no loop-carried dependencies, but there may be other properties of the loop that prevent vectorization.

In cases where vectorization is impossible, the information given in the assertion made by `vector_for` may still be put to good use in aiding other optimizations.

For more information, refer to “[#pragma vector_for](#)” on page 1-155.

#pragma all_aligned

The `all_aligned` pragma is used as shorthand for multiple `__builtin_aligned` assertions. By prefixing a `for` loop with the pragma, it is asserted that every pointer variable in the loop is aligned on a quad-word boundary at the beginning of the first iteration.

Achieving Optimal Performance from C/C++ Source Code

Therefore, adding the pragma to the following loop

```
#pragma all_aligned
for (i=0; i<100; i++)
    a[i] = b[i];
```

Good: uses `all_aligned` to inform compiler of alignment of a and b.

is equivalent to writing

```
__builtin_aligned(a, /(sharc)2/4/);
__builtin_aligned(b, /(sharc)2/4/);
for (i=0; i<100; i++)
    a[i] = b[i];
```

Good: uses `__builtin_aligned` to give alignment of a and b.

In addition, the `all_aligned` pragma may take an optional literal integer argument *n* in parentheses. This tells the compiler that all pointer variables are aligned on a quad-word boundary at the beginning of the *n*th iteration. Note that the iteration count begins at zero. Therefore,

```
#pragma all_aligned (3)
for (i=99; i>=0; i--)
    a[i] = b[i];
```

Good: uses `all_aligned` to inform compiler of alignment of a and b.

is equivalent to

```
__builtin_aligned(a+96, 4);
__builtin_aligned(b+96, 4);
for (i=99; i>=0; i--)
    a[i] = b[i];
```

Good: uses `__builtin_aligned` to give alignment of a and b.

For more information, refer to [“Using __builtin_aligned” on page 2-12.](#)

Pragmas

#pragma different_banks

The `different_banks` pragma is used as shorthand for declaring multiple pointer types with different bank qualifiers. It asserts that any two independent memory accesses in the loop may be issued together without incurring a stall.

For example,

```
#pragma different_banks
for (i=0; i<100; i++)
    a[i] = b[i] + a[i-4];
```

Good: uses different banks to allow simultaneous accesses to `a` and `b`.

which will allow a single instruction loop to be created. Note that the use of the `different_banks` pragma implies that the memory accesses do not alias each other, since if they did, it would not be permitted to issue them simultaneously. See [“#pragma different_banks” on page 1-155](#) for more information.

#pragma no_alias

When immediately preceding a loop, the `no_alias` pragma asserts that no load or store in the loop accesses the same memory as any other. This helps to produce shorter loop kernels as it permits instructions in the loop to be rearranged more freely. See [“#pragma no_alias” on page 1-156](#) for more information.

Useful Optimization Switches

Table 2-1 lists the compiler switches useful during the optimization process.

Table 2-1. C/C++ Compiler Optimization Switches

Switch Name	Description
<code>-const-read-write</code> on page 1-25	Specifies that data accessed via a pointer to <code>const</code> data may be modified elsewhere.
<code>-flags-link -e</code> on page 1-29	Specifies linker section elimination.
<code>-force-circbuf</code> on page 1-30	Treats array references of the form <code>array[i%n]</code> as circular buffer operations.
<code>-ipa</code> on page 1-33	Turns on inter-procedural optimization. Implies use of <code>-O</code> . May be used in conjunction with <code>-Os</code> or <code>-Ov</code> .
<code>-no-fp-associative</code> on page 1-37	Does not treat floating-point multiply and addition as an associative.
<code>-no-saturation</code> on page 1-37	Do not turn non-saturating operations into saturating ones.
<code>-O</code> on page 1-39	Enables code optimizations and optimizes the file for speed.
<code>-Os</code> on page 1-39	Optimizes the file for size.
<code>-Ov num</code> on page 1-40	Controls speed vs. size optimizations (sliding scale).
<code>-pguide</code> on page 1-42	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization.
<code>-save-temps</code> on page 1-45	Saves intermediate files (for example, <code>.s</code>).

Example: How the Optimizer Works

The following floating-point scalar product loop will be used to show how the optimizer works.

Example: C source code for floating-point scalar product.

```
float sp(float *a, float *b, int n) {  
    int i;  
    float sum=0;  
    for (i=0; i<n; i++) {  
        sum+=a[i]*b[i];  
    }  
    return sum;  
}
```

After code generation and conventional scalar optimizations, the compiler will have generated a loop that looks something like the following example.

Example: Initial code generated for floating-point scalar product.

```
.P1L3:  
    xR0 = [J0 += 1];;  
    xR2 = [J1 += 1];;  
    xfR4 = R0 * R2;;  
    xfR5 = R5 + R4;;  
    if n1ce0, jump  
.P1L3;;
```

The loop exit test has been moved to the bottom and the loop counter rewritten to count down to zero. This enables the generation of a hardware loop (LC0 is initialized with the loop count). sum is being accumulated in xR5. J0 and J1 hold pointers that are initialized with the parameters a and b and incremented on each iteration.

Achieving Optimal Performance from C/C++ Source Code

In order to use both compute blocks, the optimizer unrolls the loop to run two iterations in parallel. `sum` is now being accumulated in `xR5` and `yR5`, which must be added together after the loop to produce the final result. In order to use the long-word loads needed for the loop to be as efficient, the compiler has to know that `J0` and `J1` have initial values that are even. Note also that unless the compiler knows that original loop was executed an even number of times, a conditionally-executed odd iteration must be inserted outside the loop. The initial value of `LC0` has been halved.

Example: Code generated for floating-point scalar product after vectorization transformation.

```
.P1L3:
    yxR0 = 1[J0 += 2];;
    yxR2 = 1[J1 += 2];;
    xyfR4 = R0 * R2;;
    xyfR5 = R5 + R4;;
    if nlce0, jump .P1L3;;
```

If the optimizer can verify that `J0` and `J1` are initially quad-word-aligned, then it will unroll the loop to make better use of the TigerSHARC processor memory bandwidth.

In the next step, the `sum` is being calculated in `xR5`, `yR5`, `xR7` and `yR7`. `LC0` has been halved again.

Example: Code generated for floating-point scalar product after a second vectorization transformation.

```
.P1L3:
    yxR1:0 = q[J0 += 4];;
    yxR3:2 = q[J1 += 4];;
    xyfR4 = R0 * R2;;
    xyfR5 = R5 + R4;;
    xyfR6 = R1 * R3;;
    xyfR7 = R7 + R6;;
    if nlce0, jump .P1L3;;
```

Example: How the Optimizer Works

Finally, the optimizer software-pipelines the loop, unrolling and overlapping iterations to obtain highest possible use of functional units. The following code would be generated if it were known that the loop executed at least twenty times and the loop count was a multiple of eight.

Example: Code generated for floating-point scalar product after software-pipelining.

```
.P1L3:
    yxR1:0 = q[J0+=4];;
    yxR3:2 = q[J1+=4];;
    yxR9:8 = q[J0+=4];;
    xyfR4 = R0 * R2; yxR11:10 = q[J1+=4];;
    xyfR6 = R1 * R3; yxR1:0 = q[J0+=4];;
    xyfR5 = R5 + R4; xyfR12 = R8 * R10; yxR3:2 = q[J1+=4];;

.P1L28:
    xyfR7 = R7 + R6; xyfR14 = R9 * R11; yxR9:8 = q[J0+=4];
    xyfR5 = R5 + R12; xyfR4 = R0 * R2; yxR11:10 = q[J1+=4];;
    xyfR7 = R7 + R14; xyfR6 = R1 * R3; yxR1:0 = q[J0+=4];;
    if nlce0, jump .P1L28; xyfR5 = R5 + R4; xyfR12 = R8 * R10;
                                yxR3:2 = q[J1+=4];;

    xyfR7 = R7 + R6; xyfR14 = R9 * R11;;
    xyfR5 = R5 + R12; xyfR4 = R0 * R2;;
    xyfR7 = R7 + R14; xyfR6 = R1 * R3;;
    xyfR5 = R5 + R4;;
    xyfR7 = R7 + R6;;
```

Assembly Optimizer Annotations

When the compiler optimizations are enabled the compiler can perform a large number of optimizations in order to generate the resultant assembly code. The decisions taken by the compiler as to whether certain optimizations are safe or worthwhile are generally invisible to a programmer, though it could be of benefit to programmers to be given feedback from the compiler with regards to the decisions made during the optimization phase. The intention of the information provided is to give a programmer an understanding of how close to optimal a program is and what more could possibly be done to improve the generated code.

The feedback from the compiler optimizer is provided by means of annotations made to the assembly file generated by the compiler. The assembly file generated by the compiler can be kept by specifying the `-S` switch (see [on page 1-44](#)), the `-save-temps` switch (see [on page 1-45](#)) or by checking the **Project Options->Compile->General->Save temporary files** option in VisualDSP++ IDDE.

There are several areas of information provided by the assembly annotations that could be of benefit in code evaluation to improve the generated code, for example providing an indication of resource usage or the absence of a particular optimization from the resultant code, which can often be more important than its presence. The intention of the assembly code annotations is to give the programmer some insight as to the reasons why the compiler enables and disables certain optimizations for a specific code sequence.

The assembly code generated by the compiler optimizer is annotated with the following information:

- “[Procedure Statistics](#)” on [page 2-52](#)
- “[Loop Identification](#)” on [page 2-54](#)

Assembly Optimizer Annotations

- “[Vectorization](#)” on page 2-60
- “[Modulo Scheduling](#)” on page 2-66

The assembly output for the examples in this chapter may differ based on optimization flags and the version of the compiler. As a result, you may not be able to exactly reproduce these results.

Procedure Statistics

For each function call, the following is reported:

- Frame size: size of stack frame. In TigerSHARC processors, there are two stacks: $\mathbb{J}$ stack and $\mathbb{K}$ stack. Both are reported.
- Registers used. Since function calls tend to implicitly clobber registers, there are several sets:
 1. The first set is composed of the scratch registers changed by the current function, not counting the registers that are implicitly clobbered by the functions called from the current function.
 2. The second set are the call preserved registers changed by the current function, not counting the registers that are implicitly clobbered by the functions called from the current function.
 3. The third set are the registers clobbered by the inner function calls.

Example A (for ADSP-TS101 Processors)

Consider the following program:

```
struct str {  
    int x1, x2;  
};
```

Achieving Optimal Performance from C/C++ Source Code

```
int func1(struct str*, int *);
int func2(struct str s);
int foo(int in)
{
    int sum = 0;
    int local;
    struct str l_str;
    sum += func1(&l_str, &local);
    sum += func2(l_str);
    return sum;
}
```

The procedure statistics for `foo` are:

```
_foo:
//-----
//   Procedure statistics:
//
//   J Frame size           = 16 words
//   K Frame size           =  8 words
//
// Scratch registers modified:{XR4-XR5,J4-J6,J8,CJMP,jICC,kICC}
//
// Call preserved registers used:{J16-J19}
//
// Registers clobbered by function calls:
// {XR0-XR23,YR0-YR23,J0-J15,J26-J30,JB0-JB3,JL0-JL3,K0-K15,
// K26-K30,KB0-KB3,KL0-KL3,CJMP,LC0-LC1,XMR0-XMR3,YMR0-YMR3,
// XPRO-XPR1,YPRO-YPR1,XTR0-XTR15,YTR0-YTR15,XTHR0-XTHR1,
// YTHR0-YTHR1,jICC,kICC,XSTAT,YSTAT,XDAB0-XDAB3,YDAB0-YDAB3}
//-----
// "exampleA.c" line 6 col 5
J26 = J27 - 64;K26 = K27 - 64;;
[J27 += -16] = CJMP;q[K27 += -8] = J19:16;;
// " exampleA.c" line 11 col 5
J5 = J26 + 63;;
J4 = J26 + 61;;
call _func1; q[J27+4]=J27:24; q[K27+4]=K27:24;;
// " exampleA.c" line 12 col 5
```

Assembly Optimizer Annotations

```
J6 = j31 + 1;;
XR5 = [J26 + 62];;
XR4 = [J26 + 61];;
J16 = J8 + 0;;
call _func2; q[J27+4]=J27:24; q[K27+4]=K27:24;;
J8 = J8 + J16;;
CJMP = [J26 + 64];J19:16 = q[K27 + 8];;
//      -- 11 stalls --
cjmp (ABS); J27:24=q[J26+68]; K27:24=q[K26+68];;
    _foo.end:
    .global _foo;
    .type _foo,STT_FUNC;
```

Notes:

- The **J** Frame size is 16 words, and the **K** Frame size is 8 words, because this is the amount of stack space allocated by the function.
- The set of Scratch registers modified is {XR4-XR5, J4-J6, J8, CJMP, jICC, kICC} because except for the `func1` and `func2` function calls, these are the only scratch registers changed by `foo`.
- The set of Call preserved registers used is {J16-J19} because these are the only call preserved registers used by `foo`.
- The set of Registers clobbered by function calls contains the set of registers potentially changed by the calls to `func1` and `func2`.

Loop Identification

One very useful annotation is loop identification, that is, showing the relationship between the source program loops and the generated assembly code. This is not easy because due to the various loop optimizations some of the original loops vanish by being entirely unrolled, while other loops get merged, making it extremely difficult for describing what has happened to them.

Finally, the assembly code may contain compiler-generated loops that do not correspond to any loop in the user program, but rather represent constructs such as structure assignment or calls to `memcpy`.

Loop Identification Annotations

Loop identification annotation rules are:

- Annotate only the loops that originate from the C looping constructs `do`, `while`, and `for`. Therefore, any `goto` defined loop is not accounted for.
- A loop is identified by the position of the corresponding keyword (`do`, `while`, `for`) in the source file.
- Account for all such loops in the original user program.
- Generally, loop bodies are delimited between the `Lx: Loop at <file position>` and `End Loop Lx` assembly annotation. The former annotation follows the label of the first block in the loop. The later annotation follows the first jump back to the beginning of the loop. However, there are cases when the code corresponding to a user loop cannot be entirely represented between such two markers. In such cases the assembly code contains blocks that belong to a loop, but are not contained between that loop's end markers. Such blocks are annotated with a comment identifying the innermost loop they belong to `Part of Loop Lx`.
- Sometimes a loop in the original program does not show up in the assembly file, because it was either transformed or deleted. In either case, a short description of what happened to the loop is given at the beginning of the function.
- A program's innermost loops are those loops that do not contain other loops. In addition to regular loop information, the innermost loops with no control flow and no function calls are annotated with additional information such as:

Assembly Optimizer Annotations

- Cycle count: the number of cycles needed to execute one iteration of the loop, including the stalls.
- Resource usage: the resources used during one iteration of the loop. For each resource we show how many of that resource are used, how many are available and the percentage of utilization during the entire loop. Resources are shown in decreasing order of utilization. Note that 100% utilization means that the corresponding resource is used at its full capacity and represents a bottleneck for the loop. For more information on resources, see an appropriate TigerSHARC Processor programming reference.
- Some loops are subject to optimizations such as vectorization or modulo scheduling. These loops receive additional annotations as described in the vectorization and modulo scheduling paragraphs.

Example C (for ADSP-TS101 Processors)

Consider the following example:

```
1  int bar(int a[10000])
2  {
3      int i, sum = 0;
4      for (i = 0; i < 9999; ++i)
5          sum += (sum + 1);
6      while (i-- < 9999) /* this loop doesn't get executed */
7          a[i] = 2*i;
8      return sum;
9  }
```

The two loops are accounted for as follows:

```
_bar:
//-----
..... procedure statistics .....
//-----
// Original Loop at " exampleC.c" line 6 col 5 -- loop structure
   removed due to constant propagation.
```

Achieving Optimal Performance from C/C++ Source Code

```
//-----  
//" exampleC.c" line 1 col 5  
    YR0 = 0;;  
//" exampleC.c" line 4 col 5  
    LC0 = 9999;;  
.P1L1:  
//-----  
// Loop at " exampleC.c" line 4 col 5  
//-----  
// This loop executes 1 iteration of the original loop in 4 cycles.  
// (cycle count 4 includes 2 stalls)  
//-----  
// This loop's resource usage is:  
//    Y Compute Block ALU    used    2 out of    4 ( 50.0%)  
//    Y Compute Block        used    2 out of    8 ( 25.0%)  
//-----  
//" exampleC.c" line 5 col 2  
//    -- stall --  
    YR1 = INC R0;;  
//    -- stall --  
    if nlc0e, jump .P1L1 ;YR0 = R0 + R1;;  
//-----  
// End Loop L1  
//-----  
//-----  
// Part of top level (no loop)  
//-----  
//" exampleC.c" line 8 col 5  
    J8 = YR0;;  
    cjmp (ABS);;  
  
_bar.end:
```

Notes:

- The keywords identifying the two loops are:
 1. `for`—whose position is in the file `loop_id.c`, line 4, column 5
 2. `while`—whose position is in the file `loop_id.c`, line 6, column 5

Assembly Optimizer Annotations

- Immediately after the procedure statistics, there is a message stating that the loop at line 6 in the user program was removed. The reason was constant propagation, which in this case realizes that the value of `I` after the first loop is 9999 and the second loop does not get executed.
- The start of the loop at line 4 is marked in the assembly by the ‘Loop at loop_id.c, line 4, column 5’ annotation. This annotation follows the loop label `.P1L1`. The loop label “End Loop L1” is used to identify the end of the loop.
- The loop resource information accounts for all instructions and stalls inside the loop. In our case, the loop body is executed in four cycles (2 stalls and 2 instruction lines). During these four cycles the Y ALU resource could have been used 4 times, but it is only used 2 times, yielding a 50% utilization. Note that the first stall is caused by the dependence between the write to `YR0` in the last line and the use of `YR0` in the next iteration of the loop.

File Position

As seen in the Example C, a file position is given using the file name, line number and the column number in that file: "loop_id.c" line 4 col 5.

This scheme uniquely identifies a source code position, unless inlining is involved. In presence of inlining, a piece of code from a certain file position can be inlined at several places, which in turn can be inlined at other places. Since inlining can happen for an unspecified number of times, a recursive scheme is used to describe a general file position.

Therefore, a <general file position> is <file position> inlined from <general file position>.

Example D (Inlining)

Consider the following source code:

```
5  void f2(int n);
6  inline void f3(int n)
7  {
8      while(n--)
9          f4();
10         if (n == 7)
11             f2(3*n);
12 }
13
14 inline void f2(int n)
15 {
16     while(n--) {17  f3(n);
18         f3(2*n);
19     }
20 }
21 void f1(volatile unsigned int i)
22 {
23     f2(30);
24 }
```

Here is some of the code generated for function f1:

```
_f1:
.P2L1:
//-----
// Loop at "exampleD.c" line 16 col 5 inlined from
// "exampleD.c" line 23 col 7
//-----
// "exampleD.c" line 8 col 5
YR0 = PASS R24; YR29 = YR27;;
if yaeq, jump .P2L2 (NP); else, YR30 = YR27;
else, YR31 = LSHIFT R28 BY 0;;

.P2L4:
//-----
// Loop at "exampleD.c" line 8 col 5 inlined from
// "exampleD.c" line 17 col 4 inlined from "exampleD.c"
// line 23 col 7
```

Assembly Optimizer Annotations

```
//-----  
//-----  
// End Loop L4  
//-----  
  
...  
  
.P2L9:  
//-----  
// Loop at " exampleD.c" line 8 col 5 inlined from "exampleD.c"  
// line 18 col 4 inlined from "exampleD.c" line 23 col 7  
//-----  
  
//-----  
// End Loop L9  
//-----  
// End Loop L1
```

Vectorization

The trip count of a loop is the number of times the loop body gets executed.

Under certain conditions, the compiler is able to take two operations executed in consecutive iterations of a loop and to execute them in only one more powerful instruction giving a loop with a smaller trip count. The transformation in which operations from two subsequent iterations are executed in one more powerful single operation is called vectorization.

For instance, the original loop may start with a trip count of 1000.

```
for(i=0; i< 1000; ++i)  
    a[i] = b[i] + c[i];
```

and, after the optimization, end up with the vectorized loop with a final trip count of 500. The vectorization factor is the number of operations in the original loop that are executed at once in the transformed loop. It is illustrated using some pseudo code below.

```
for(i=0; i< 500; i+=2)
```

Achieving Optimal Performance from C/C++ Source Code

```
(a[i], a[i+1]) = (b[i],b[i+1]) .plus2. (c[i], c[i+1]);
```

In the above example, the vectorization factor is 2. A loop may be vectorized more than once.

If the trip count is not a multiple of the vectorization factor, some iterations need to be peeled off and executed unvectorized. Thus, if in the previous example, the trip count of the original loop was 1001, then the vectorized code would be:

```
for(i=0; i< 500; i+=2)
    (a[i], a[i+1]) = (b[i],b[i+1]) .plus2. (c[i], c[i+1]);
a[1000] = b[1000] + c[1000];
// This is one iteration peeled from
// the back of the loop.
```

In the above examples, the trip count is known and the amount of peeling is also known. If the trip count is not known (it is a variable), the number of peeled iterations depends on the trip count, and in such cases, the optimized code contains peeled iterations that are executed conditionally.

Unroll and Jam

Another vectorization related transformation is unroll and jam. Let's consider the following function:

```
/* unroll and jam example */
void f_unroll_and_jam(int a[][40], int *restrict c) {
    int i, j;
    for (i=0; i<60; i++) {
        int sum=0;
        for (j=0; j<40; j++) {
            sum += a[j][i];
        }
        c[i] = sum;    }
}
```

Assembly Optimizer Annotations

The outer loop can be unrolled twice and the result is:

```
void f_unroll_and_jam(int a[][40], int *restrict c) {
    int i, j;
    for (i=0; i<30; i+=2) {
        {
            int sum=0;
            for (j=0; j<40; j++) {
                sum += a[j][i];
            }
            c[i] = sum;
        }
        {
            int sum=0;
            for (j=0; j<40; j++) {
                sum += a[j][i+1];
            }
            c[i+1] = sum;
        }
    }
}
```

The two inner loops can be jammed together. We shall assume that we have a `plus_eq2` operation which is a more powerful version of `+=` that can handle two integers at a time. The result is:

```
void f_unroll_and_jam(int a[][40], int *restrict c) {
    int i, j;
    for (i=0; i<30; i+=2) {
        int sum0=0;
        int sum1=0;
        for (j=0; j<40; j++) {
            (sum0, sum1).plus_eq2. (a[j][i], a[j][i+1]);
        }
        (c[i], c[i+1]) = (sum0, sum1);
    }
}
```

The above sequence of transformation, where an outer loop is unrolled and the copies corresponding to the inner loops are jammed together is called unroll and jam.

Assembly Code Unroll and Jam Example:

The assembly annotated code for the above `f_unroll_and_jam` example is:

```
.P1L1:
//-----
//  Loop at "f_unroll_and_jam.c" line 3 col 3
//-----
//  Loop was unrolled for unroll and jam 2 times
//-----
..... some outer loop code .....
.P1L10:
//-----
//  Loop at "f_unroll_and_jam.c" line 5 col 5
..... other loop annotations .....
//-----
//  Loop was jammed by unroll and jam 2 times
//-----
..... jammed loop code .....
//-----
//  End Kernel for Loop L10
//-----

.P1L11:
..... more outer loop code .....
    if nlc0e, jump .P1L1 ; [J5 + -1] = YR0;;
//-----
//  End Loop L1
//-----
```

Loop Flattening

Another transformation, slightly related to vectorization, is loop flattening. The loop flattening operation takes two nested loops that run N_1 and N_2 times, respectively, and transforms them into a single loop that runs $N_1 \times N_2$ times. For instance, the following function

```
void copy_v(int a[][100], int b[][100]) {
    int i,j;
    for (i=0; i< 30; ++i)
        for (j=0; j < 100; ++j)
            a[i][j] = b[i][j];
}
```

Assembly Optimizer Annotations

is transformed into

```
void copy_v(int a[][100], int b[][100]) {
    int i,j;
    int *p_a = &a[0][0];
    int *p_b = &b[0][0];
    for (i=0; i< 3000; ++i)
        p_a[i] = p_b[i];
}
```

This may further facilitate the vectorization process:

```
void copy_v(int a[][100], int b[][100]) {
    int i,j;
    int *p_a = &a[0][0];
    int *p_b = &b[0][0];
    for (i=0; i< 3000; i+=2)
        (p_a[i], p_a[i+1]) = (p_b[i], p_b[i+1]);
}
```

Vectorization Annotations

For every loop that is vectorized, the following information is provided:

- The vectorization factor
- The number of peeled iterations
- The position of the peeled iterations (front or back of the loop)
- Information about whether peeled iterations are conditionally or unconditionally executed

For every loop pair that is subject to unroll and jam, it is necessary to:

- Annotate the unrolled outer loop with the number of times it was unrolled
- Annotate the inner loop with the number of times the loop was jammed

For every loop pair that is subject to loop flattening, it is necessary to account for the loop that is lost and show the remaining loop that it was merged with.

Example E (Vectorization for ADSP-TS101 processors):

Consider the test program:

```
void add(int *a, int *restrict b, int *restrict c, int dim) {
    int i;
    for (i = 0 ; i < dim; ++i)
        a[i] = b[i] + c[i];
}
```

for which the vectorization information is:

```
.P1L24:
//-----
// Loop at "exampleE.c" line 3 col 5
..... other loop annotations .....
//-----
// Loop was vectorized by a factor of 4.
//-----
// Vectorization peeled 3 conditional iterations from the back
// of the loop because of an unknown trip count, possible
// not a multiple of 4.
//-----
..... loop body .....
    if nlc0e, jump .P1L24 ; [J4 + -1] = YR9 ; YR1:0 = XR3:2;;
//-----
// End Kernel for Loop L24
//-----
```

In this example, the vectorization factor is 4. Since the trip count `dim` is unknown, three conditional iterations are peeled from the back of the loop, corresponding to the three cases when `dim` is $4k+1$, $4k+2$ or $4k+3$. The loop end is marked with `End Kernel`, because in this case the loop was modulo scheduled (see [“Modulo Scheduling” on page 2-66](#)).

Assembly Optimizer Annotations

Loop Flattening Example:

The assembly annotations for the `copy_v` function (from “[Loop Flattening](#)” on page 2-63) are:

```
_copy_v:
//-----
..... procedure statistics .....
//-----
// Original Loop at "test_flatten_loop_dim.c" line 3 col 5 --
// loop flatten into Loop at "test_flatten_loop_dim.c" line 4 col 2
//-----
..... procedure code .....
.P1L1:
//-----
// Loop at "test_flatten_loop_dim.c" line 4 col 2
//-----
..... loop annotations .....
..... loop body .....
    if nlc0e, jump .P1L1 ; [J4 + -1] = XR3;;
//-----
// End Loop L1
//-----
```

Modulo Scheduling

Modulo scheduling is a kind of software pipelining. It is used to schedule innermost loops without control flow. There are various algorithms for modulo scheduling, and all of them produce scheduled loops that can be described by the following parameters:

- Initiation Interval (II): the number of cycles between initiating two successive iterations of the loop
- Stage Count (SC): the number of initiation intervals until the first iteration of the loop has completed. This is also the number of iterations in progress at any time within the kernel.

- Modulo Variable Expansion unroll factor (MVE unroll): the number of times the loop has to be unrolled to achieve the schedule without overlapping register lifetimes
- Trip count: the number of times the loop goes around
- Trip modulo: a number that is known to divide the trip count
- Trip maximum: an upper limit for the trip count
- Trip minimum: a lower limit for the trip count
- Minimum initiation interval due to resources (res_MII): a lower limit for the Initiation interval (II), imposed by the fact that at least one of the resources is used at maximum capacity.

The original loop instructions end up in three places: the prolog, the kernel and the epilog of the scheduled loop. The kernel is the most important and it is the body of the scheduled loop. The prolog and the epilog contain instructions peeled from the original loop that precede and follow the kernel, respectively.

Modulo Scheduling Annotations:

For every modulo scheduled loop, in addition to regular loop annotations, the following information is provided:

- The initiation interval, II
- The final trip count if it is known: the trip count of the loop as it ends up in the assembly code
- A cycle count representing the time to run one iteration of the pipelined loop
- The minimum trip count, if it is known and the trip count is unknown

Assembly Optimizer Annotations

- The maximum trip count, if it is known and the trip count is unknown
- The trip modulo, if it is known and the trip count is unknown
- The stage count (iterations in parallel)
- The MVE unroll factor
- The resource usage
- The Minimum initiation interval due to resources (`res MII`)

In addition to the above modulo scheduled loop annotations, the compiler can optionally produce more annotation information for the instructions that belong to the prolog, kernel or the epilog of the modulo scheduled loop. In this case, the instructions are annotated with the following information:

- The loop label the instruction is related to
- The part of the modulo scheduled loop (prolog, kernel or epilog) that the instruction belongs to
- ID: a unique number associated with the original instruction in the unscheduled loop that generates the current instruction. It is useful, because a single instruction in the original loop can expand into multiple instructions in a modulo scheduled loop.

The ID are assigned in the order the instructions appear in the kernel (though they might repeat for MVE unroll > 1).

- `sn`: the stage count the instruction belongs to. If the loop has a stage count of 1, the instruction's `sn` is not shown.
- `rs`: the register set used for the current instruction (useful when MVE unroll > 1, in which case `rs` can be `0, 1, ..., mve-1`). If the loop has an MVE of 1, the instruction's `rs` is not shown.

Achieving Optimal Performance from C/C++ Source Code

- In addition to the above, the instructions in the kernel are annotated with:
 - `Iter`: specifies what iteration of the original loop an instruction is on in the schedule.
 - In a modulo scheduled kernel, there are instructions from $(SC+MVE-1)$ iterations in the original loop. Each of them can be chosen as “the current” iterations, relative to which the other iterations are specified. Chose `Iter=0` for the instructions in the earliest iterations of the original loop.

If the annotations for modulo scheduled instructions are enabled, then the format of an assembly line containing several instructions is changed from:

```
instruction_1; instruction_2; instruction_3;;
```

to:

```
/**/ instruction_1;  
      instruction_2;  
      instruction_3;;
```

This is done to allow for annotations at the instruction level:

```
/**/ instruction_1;    // {annotations for instruction_1}  
      instruction_2;    // {annotations for instruction_2}  
      instruction_3;;  // {annotations for instruction_3}
```

The `/**/` marks the beginning of an instruction line.

Example -- Modulo Scheduling Annotations:

Consider the following function:

```
1  void add(int *a, int *restrict b, int *restrict c) {  
2      int i;  
3      for (i = 0; i < 200; ++i)
```

Assembly Optimizer Annotations

```
4          a[i] = b[i] + c[i];
5      }
```

The annotated output for the modulo scheduling, including the optional annotation of the modulo scheduling related instructions, is:

```
8  _add:
9  //-----
10 // Procedure statistics:
11 //
12 //   J Frame size          = 12 words
13 //   K Frame size          =  4 words
14 //
15 // Scratch registers modified:
16 //   {XR0-XR11,YR0-YR5,YR8-YR9,J0-J1,J4,
17 //     JB0,JL0,LC0,xACC,yACC,jICC,kICC}
18 //
19 // No call preserved registers used.
20 //-----
21 // test_mod_sched.c" line 1 col 6
22 /**/ J26 = J27 - 64;
23      K26 = K27 - 64;;
24 /**/ J27 = J27 - 12;
25      K27 = K27 - 4;;
26 /**/ JB0 = 0;
27      J0 = J6 + 0;;
28 /**/ JL0 = 0;
29      J1 = J5 + 0;;
30 /**/ YR3:0 = DAB q[J1 += 4];
31      LC0 = 25;;
32 /**/ YR3:0 = DAB q[J1 += 4];;
33      // {L7 prolog:id=3,sn=0,rs=0}
34 /**/ XR3:2 = YR1:0;;    // {L7 prolog:id=6,sn=0,rs=0}
35 //   -- stall --
36 /**/ XR7:4 = DAB q[J0 += 4];;
37 /**/ XR7:4 = DAB q[J0 += 4];;
38      // {L7 prolog:id=1,sn=0,rs=0}
39 /**/ YR5:4 = XR7:6;;    // {L7 prolog:id=9,sn=0,rs=0}
40
41 .P1L7:
42 //-----
43 // Loop at "test_mod_sched.c" line 3 col 5
```

Achieving Optimal Performance from C/C++ Source Code

```
40 //
41 // This loop executes 8 iterations of the original loop
   // in 12 cycles.
42 //-----
43 // Trip Count = 24
44 //-----
45 // Successfully found modulo schedule with:
46 // Initiation Interval (II) = 6
47 // Stage Count (SC) = 2
48 // MVE Unroll Factor = 2
49 // Minimum initiation interval due to resources
   // (res MII) = 6.00
50 //-----
51 // This loop's resource usage is:
52 //
53 //
54 //
55 //          resource statistics
56 //
57 //
58 //
59 //
60 //
61 //-----
62 // Loop was vectorized by a factor of 4.
63 //-----
64 //          /**/ XR11:8 = DAB q[J0 += 4];;
   //          // {L7 kernel:id=1,sn=0,rs=1,Iter=1}
65 // "test_mod_sched.c" line 4 col 2
66 //          /**/ xyR5:4 = R5:4 + R3:2;
   //          // {L7 kernel:id=2,sn=1,rs=0,Iter=0}
67 //          YR3:0 = DAB q[J1 += 4];;
   //          // {L7 kernel:id=3,sn=0,rs=1,Iter=1}
68 //          /**/ [J4 += 4] = XR4;;
   //          // {L7 kernel:id=4,sn=1,rs=0,Iter=0}
69 //          /**/ [J4 + -3] = XR5;
   //          // {L7 kernel:id=5,sn=1,rs=0,Iter=0}
70 //          XR3:2 = YR1:0;;
   //          // {L7 kernel:id=6,sn=0,rs=1,Iter=1}
71 //          /**/ [J4 + -2] = YR4;;
   //          // {L7 kernel:id=7,sn=1,rs=0,Iter=0}
```

Assembly Optimizer Annotations

```
72      /**/ if lc0e, jump .P1L8 (NP);
           // {L7 kernel:id=10,sn=0,rs=1,Iter=1}
73      [J4 + -1] = YR5;
           // {L7 kernel:id=8,sn=1,rs=0,Iter=0}
74      YR9:8 = XR11:10;;
           // {L7 kernel:id=9,sn=0,rs=1,Iter=1}
75      /**/ XR7:4 = DAB q[J0 += 4];;
           // {L7 kernel:id=1,sn=0,rs=0,Iter=2}
76      /**/ xyR9:8 = R9:8 + R3:2;
           // {L7 kernel:id=2,sn=1,rs=1,Iter=1}
77      YR3:0 = DAB q[J1 += 4];;
           // {L7 kernel:id=3,sn=0,rs=0,Iter=2}
78      /**/ [J4 += 4] = XR8;;
           // {L7 kernel:id=4,sn=1,rs=1,Iter=1}
79      /**/ [J4 + -3] = XR9;
           // {L7 kernel:id=5,sn=1,rs=1,Iter=1}
80      XR3:2 = YR1:0;;
           // {L7 kernel:id=6,sn=0,rs=0,Iter=2}
81      /**/ [J4 + -2] = YR8;;
           // {L7 kernel:id=7,sn=1,rs=1,Iter=1}
82      /**/ jump .P1L7 ;
83      [J4 + -1] = YR9;
           // {L7 kernel:id=8,sn=1,rs=1,Iter=1}
84      YR5:4 = XR7:6;;
           // {L7 kernel:id=9,sn=0,rs=0,Iter=2}
85      //-----
86      // End Kernel for Loop L7
87      //-----
88
89      .P1L8:
90      /**/ xyR1:0 = R9:8 + R3:2;;
           // {L7 epillog:id=2,sn=1,rs=1}
91      /**/ [J4 += 4] = XR0;;
           // {L7 epillog:id=4,sn=1,rs=1}
92      /**/ [J4 + -1] = YR1;;    // {L7 epillog:id=8,sn=1,rs=1}
93      /**/ [J4 + -2] = YR0;;    // {L7 epillog:id=7,sn=1,rs=1}
94      /**/ [J4 + -3] = XR1;;    // {L7 epillog:id=5,sn=1,rs=1}
95      /**/ cjmp (ABS); J27:24=q[J26+68]; K27:24=q[K26+68];;
96
97      ._add.end:
98      .global _add;
99      .type _add,STT_FUNC;
```

Achieving Optimal Performance from C/C++ Source Code

Notes:

Lines 43-60 define the kernel information: loop name and modulo schedule parameters: `II`, stage count, etc.

Lines 64-84 show the kernel.

Each instruction in the kernel has an annotation between `{}`, inside a comment following the instruction. If several instructions are executed in parallel, each gets its own annotation.

For instance, line 64 looks like:

```
64    /**/ XR11:8 = DAB q[J0 += 4];;
        // {L7 kernel:id=1,sn=0,rs=1,Iter=1}
```

This annotation describes:

- The fact that this instruction belongs to the kernel of the loop starting at `L7`.
- `ID=1` shows that this and the other two instructions that have `ID=1` originate from the same original instruction in the unscheduled loop:

```
34    /**/ XR7:4 = DAB q[J0 += 4];;
        // {L7 prolog:id=1,sn=0,rs=0}
64    /**/ XR11:8 = DAB q[J0 += 4];;
        // {L7 kernel:id=1,sn=0,rs=1,Iter=1}
75    /**/ XR7:4 = DAB q[J0 += 4];;
        // {L7 kernel:id=1,sn=0,rs=0,Iter=2}
```

- `sn=0` shows that this instruction belongs to stage count 0.
- `rs=1` shows that this instruction uses register set 1.
- `Iter=1` specifies that this instruction belongs to the second iteration of the original loop (`Iter` numbers are zero-based).

Assembly Optimizer Annotations

The prolog and epilog are not clearly delimited in blocks by themselves, but their corresponding instructions are annotated similar to the ones in the kernel except that they do not have an 'Iter' field and that they are preceded by a tag specifying whose loop prolog or epilog they belong to.

Examples:

```
30      /**/ YR3:0 = DAB q[J1 += 4];;  
          // {L7 prolog:id=3,sn=0,rs=0}  
90      /**/ xyR1:0 = R9:8 + R3:2;;  
          // {L7 epilog:id=2,sn=1,rs=1}
```

Note that the prolog/epilog instructions may mix with other instructions on the same line.

This situation does not occur in this example, but in a different example it might have:

```
      /**/ xyR0 = R0 + R3;    // {L10 epilog:id=2,sn=1,rs=1}  
      J4 = J4 + 2;;
```

This shows a line with two instructions. The second instruction "J4 = J4 + 2" is unrelated to modulo scheduling, and therefore it has no annotation.

3 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 3-3](#)
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 `ccts` compiler.
- [“Documented Library Functions” on page 3-37](#)
It lists the functions (defined by the C standard header files) that are described in [“Run-Time Library Reference” on page 3-41](#).
- [“Run-Time Library Reference” on page 3-41](#)
It provides reference information about the C run-time library functions included with this release of the `ccts` compiler.

The `ccts` compiler provides a broad collection of library functions, including those required by the ANSI standard and additional functions supplied by Analog Devices, Inc. that are of value in signal processing. 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 as well as 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 (ASIN: 0138220646). 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 and provides information on the following topics:

- [“Calling Library Functions” on page 3-3](#)
- [“Using Compiler’s Built-In C Library Functions” on page 3-4](#)
- [“Linking Library Functions” on page 3-5](#)
- [“Working With Library Source Code” on page 3-7](#)
- [“Working With Library Header Files” on page 3-8](#)
- [“Abridged C++ Library Support” on page 3-30](#)

The C run-time libraries’ functions, available in this `ccts` compiler release are listed in [“Documented Library Functions” on page 3-37](#). For information on the Abridged C++ library’s contents, see [“Abridged C++ Library Support” on page 3-30](#) 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 [“Run-Time Library Reference” on page 3-41](#) 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 3-7](#).

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 1-213](#).



You can use `elfar` (the archiver), described in the *VisualDSP++ 3.5 Linker and Utilities Manual for 32-Bit Processors*, to build library archive files of your own functions.

Using 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. Built-in functions are defined for some ANSI C `math.h`, `string.h`, and `stdlib.h` functions. There are also built-in functions to support various Analog Devices' extensions to the ANSI standard defined in the include file `builtns.h`. Not all built-in functions have a library alternate definition. Therefore, the failure to include the required header files may 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 3-1](#). The libraries and start-up files are installed within the subdirectory `...\TS\lib` of your VisualDSP++ installation.

Table 3-1. C and C++ Library Files

TSlib Directory	Description
libc_TS101.dlb libc_TS201.dlb	C run-time library
libcpp_TS101.dlb libcpp_TS201.dlb	C++ run-time library
libcpp_TS101_x.dlb libcpp_TS201_x.dlb	C++ run-time library with exception handling
libcpprt_TS101.dlb libcpprt_TS201.dlb	C++ run-time support library
libcpprt_TS101_x.dlb libcpprt_TS201_x.dlb	C++ run-time support library with exception handling
libdsp_TS101.dlb libdsp_TS201.dlb	DSP run-time library
libio_TS101.dlb libio_TS201.dlb	I/O run-time library
libsim.dlb libsim_TS201.dlb	Simulator library support
libx_TS101.dlb libx_TS201.dlb	C++ exception handling support library
ts_hdr_TS101.doj ts_hdr_TS201.doj	C start-up file -- calls setup routine and <code>main()</code>
ts_hdr_cpp_TS101.doj ts_hdr_cpp_TS201.doj	C++ start-up file -- calls setup routine and <code>main()</code>
ts_exit_TS101.doj ts_exit_TS201.doj	C exit routine

C and C++ Run-Time Libraries Guide

Table 3-1. C and C++ Library Files (Cont'd)

TSlib Directory	Description
ts_exit_cpp_TS101.doj ts_exit_cpp_TS201.doj	C++ exit routine
meminit_TS101.doj meminit_TS201.doj	Memory initialization support

In general, several versions of the libraries and start-up files are supplied in binary form. For instance, the DSP run-time library is built twice; once for the ADSP-TS101 processor and once for the ADSP-TS20x processors.

Binary files that are built for the ADSP-TS101 processor have a `_TS101` suffix and binary files built for the ADSP-TS20x processors have a `_TS201` suffix.

Other variants of the library and start-up files are supplied that are compatible with applications that have been compiled with the compiler switch `-char-size-8` ([on page 1-25](#)); these binary files have a `_ba` suffix. File names that do not have a `_ba` suffix have been built for applications that use the conventional, word-addressing, mode.

Binary files that have a `_mt` suffix have been built for multi-threaded environments; file names that do not have a `_mt` suffix have been built for a single-threaded environment.

Versions of the binary files are also provided that avoid a hardware anomaly that may only affect the ADSP-TS101 processor and is associated with the Branch Target Buffer (BTB). These files have been built with the compiler switch `-default-branch-np` and have a `_np` suffix.

The default libraries and start-up files for the TS20x processors have been built with all available workarounds applicable to the TS20x silicon. The file name of these binary files have a `_w` suffix. An alternative set of binary files is available that does not incorporate workarounds for any known

hardware errata. These files do not have a `_w` suffix in their file names, and are linked into an application that is built with the `-si-revision none` compiler switch (on page 1-45).

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-TS<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_TS201 -ldsp_TS201` will add the libraries `libc_TS201.dlb` and `libdsp_TS201.dlb` to the list of libraries to be searched by the linker. For more information on the `.LDF` file, see the *VisualDSP++ 3.5 Linker and Utilities Manual for 32-Bit Processors*.

 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 1-29), which ensures that that the C++ libraries are linked from the default `.LDF` files. For example, `-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 and DSP run-time libraries 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 `... \TS\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 code as a 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 the TigerSHARC assembly language and an understanding of the run-time environment, as explained in [“C/C++ Run-Time Model and Environment” on page 1-185](#). 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.

The header files define the non-suffixed names (for example, `sin`) as the 32-bit versions (for example, `sinf`) if the compiler is treating doubles as 32 bits or as the 64-bit version (for example, `sind`). This lets you use the non-suffixed names with arguments of type `double`, regardless of whether doubles are 32 or 64 bits.

The routines suffixed with `f` always require 32-bit arguments, and the routines suffixed with `d` always require 64-bit arguments, regardless of whether the `double` type is 32 or 64 bits.

Some of these routines set the `errno` global variable, as required by the C standard. To use `errno`, you must include the header file `errno.h`.

By default, the `ccts` compiler makes the `double` type 32 bits, for high performance. You can select a compiler option to make the `double` type 64 bits for conformance with the C standard, but this makes computations with double values slower. In either case, the `float` type is 32 bits, and the `long double` type is 64 bits.

A list of the header files that are supplied with this release of the `ccts` compiler appears in [Table 3-2](#). You should use a C standard text to augment the information supplied in this chapter.

Table 3-2. C Run-Time Library Header Files

Header	Purpose	Standard
<code>assert.h</code>	Diagnostics	ANSI
<code>ctype.h</code>	Character Handling	ANSI
<code>errno.h</code>	Error Handling	ANSI
<code>float.h</code>	Floating Point	ANSI
<code>iso646.h</code>	Boolean Operators	ANSI
<code>limits.h</code>	Limits	ANSI
<code>locale.h</code>	Localization	ANSI
<code>math.h</code>	Mathematics	ANSI
<code>setjmp.h</code>	Non-Local Jumps	ANSI
<code>signal.h</code>	Signal Handling	ANSI
<code>stdarg.h</code>	Variable Arguments	ANSI
<code>stddef.h</code>	Standard Definitions	ANSI
<code>stdio.h</code>	Input/Output	ANSI
<code>stdlib.h</code>	Standard Library	ANSI

C and C++ Run-Time Libraries Guide

Table 3-2. C Run-Time Library Header Files (Cont'd)

Header	Purpose	Standard
<code>string.h</code>	String Handling	ANSI
<code>time.h</code>	Time and data handling	ANSI

The following are the descriptions of the header files contained in the C library. The header files are listed in alphabetical order.

`assert.h`

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

`ctype.h`

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

`errno.h`

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

`float.h`

The `float.h` header file contains definitions of size and precision values for each C floating point data type. The `FLT_ROUNDS` macro defined in the header file is set to the C run-time environment definition of the rounding mode for `float` variables which is round-toward-nearest. The rounding mode for `long double` variables is truncation.

iso646.h

The `iso646.h` header file defines symbolic names for certain C operators; the symbolic names and their associated value are shown in [Table 3-3](#).

Table 3-3. Symbolic Names Defined in `iso646.h`

Symbolic Name	Equivalent
<code>and</code>	<code>&&</code>
<code>and_eq</code>	<code>&=</code>
<code>bitand</code>	<code>&</code>
<code>bitor</code>	<code> </code>
<code>compl</code>	<code>~</code>
<code>not</code>	<code>!</code>
<code>not_eq</code>	<code>!=</code>
<code>or</code>	<code> </code>
<code>or_eq</code>	<code> =</code>
<code>xor</code>	<code>^</code>
<code>xor_eq</code>	<code>^=</code>



The symbolic names have the same name as the C++ keywords that are accepted by the compiler when the `-alttok` switch (see [on page 1-23](#)) is specified.

limits.h

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

locale.h

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

math.h

The `math.h` header includes trigonometric, power, logarithmic, exponential, and other miscellaneous functions. The library contains the functions specified by the C standard along with implementations for the data types `float` and `long double`.

For every function that is defined to return a `double`, the `math.h` header file also defines corresponding functions that return a `float` and a `long double`. The names of the `float` functions are the same as the equivalent `double` function with `f` appended to its name. Similarly, the names of the `long double` functions are the same as the `double` function with `d` appended to its name. For example, the header file contains the following prototypes for the sine function:

```
float sinf (float x);
double sin (double x);
long double sind (long double x);
```

When the compiler is treating `double` as 32 bits, the header file will arrange that all references to the `double` functions are directed to the equivalent `float` function (with the suffix `f`). This allows you use the un-suffixed names with arguments of type `double`, regardless of whether doubles are 32 or 64 bits long.

This header file also provides prototypes for a number of additional math functions provided by Analog Devices, such as `favg`, `fmax`, `fclip`, and `copysign`.

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`.

A number of functions (including `fabs`, `favg`, `fmax`, `fmin`, `fclip`, and `copysign`) are implemented via intrinsics (provided the header file has been `#include 'd`) that map to a single machine instruction.



If the header is not included, the library implementation is used instead, at a considerable loss in efficiency.

Individual function descriptions focus on the 32-bit float version. When the full range is given for **Domain**, the 64-bit `float` interfaces are not limited to 3.4×10^{38} .

setjmp.h

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

signal.h

The `signal.h` header file provides function prototypes for the standard ANSI `signal.h` routines and also for several TigerSHARC processor extensions, such as `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.h

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

stddef.h

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

stdio.h

The `stdio.h` header file defines a set of functions, macros, and data types for performing input and output. Applications that use the facilities of this header file should link with the I/O library in the same way as linking with the C run-time library (see [“Linking Library Functions” on page 3-5](#)). The library is thread-safe but it is not interrupt-safe and should not therefore be called either directly or indirectly from an interrupt service routine.

The compiler uses definitions within the header file to select an appropriate set of functions that correspond to the currently selected addressing mode (either word-addressing or byte-addressing) and size of type `double` (either 32 bits or 64 bits). Any source file that uses the facilities of `stdio.h` must therefore include the header file. Failure to include the header file will result in a linker failure as the compiler must see a correct function prototype in order to generate the correct calling sequence.

The implementation of the `stdio.h` routines is based on a simple interface with a device driver that provides a set of low-level primitives for `open`, `close`, `read`, `write`, and `seek` operations. The device driver that is used by default bases these low-level primitives on services supported by the VisualDSP++ simulator and EZ-KIT Lite systems and which provide access to files on the host system. Alternative device drivers may be registered that can then be used transparently through the `stdio.h` functions. See [“Extending I/O Support To New Devices” on page 1-198](#) for a description of the feature.

The following restrictions apply to this software release:

- the functions `tmpfile` and `tmpnam` are not available
- the functions `rename` and `remove` are only supported under the default device driver supplied by the VisualDSP++ simulator and EZ-KIT Lite systems, and they only operate on the host file system
- positioning within a file that has been opened as a text stream is only supported if the lines within the file are terminated by the character sequence `\r\n`

At program termination, the host environment will close down any physical connection between the application and an opened file. However, the I/O library will not implicitly close any opened streams to avoid unnecessary overheads (particularly with respect to memory occupancy). Thus, unless explicit action is taken by an application, any unflushed output may be lost. Any output generated by `printf` is always flushed but output generated by other library functions, such as `putchar`, `fwrite`, `fprintf`, will not be automatically flushed. Applications should therefore arrange to close down any streams that they open. Note that the function reference `fflush(NULL)`; will flush the buffers of all opened streams.



Each opened stream is allocated a buffer which either contains data from an input file or output from a program. For text streams, this data is held in the form of 8-bit characters that are packed into 32-bit memory locations. Due to internal mechanisms used to

unpack and pack this data, the buffer must not reside at a memory location that is greater than the address 0x3fffffff. Since the `stdio` library allocates buffers from the heap, this restriction implies that the heap should not be placed at address 0x40000000 or above.

The restriction may be avoided by using the `setvbuf` function to allocate the buffer from alternative memory, as in the following example.

```
#include <stdio.h>

char buffer[BUFSIZ];
setvbuf(stdout,buffer,_IOLBF,BUFSIZ);
printf("Hello World\n");
```

This example assumes that the buffer will reside at a memory location that is less than 0x40000000.

When using the default device driver, all I/O operations are channeled through the C function `_primIO()`. The assembly label has two underscores, `__primIO`. Upon entry to `_primIO()`, the data for the request will reside in a control block at the label `_primIOCB`. The host arranges to intercept control when it enters the routine, and, after servicing the request, returns control to the calling routine. See [“Default Device Driver Interface” on page 3-17](#) for a more detailed description.

A faster set of functions is available for applications that print only to standard output. These functions are linked into an application if you compile with the switch `-flags-link -MD__USING_LIBSIM=1`. This switch forces the linker to link against the library `libsim.dlb`. This library contains a limited set of `stdio.h` facilities that are executed on the host of the VisualDSP++ debugger rather than inside the debugger’s target. This type of execution leads to smaller applications and faster output.

The following functions are supported by the `libsim.dlb` library for this compiler release only.

```
printf
sprintf
fprintf
```

All three of these functions from `libsim.dlb` use the `L` modifier, as in `%LF`, to specify a conversion for a `long double` (64-bit floating-point) argument. They also use the `"ll"` modifier, as in `%lld`, to specify a conversion for a `long long` (64-bit integer) argument. The `fprintf` routine currently ignores its `FILE*` stream argument and always prints to standard output.

Default Device Driver Interface

The `stdio` functions provide access to the files on a host system through a device driver that supports a set of low level I/O primitives. These low level primitives are described under [“Extending I/O Support To New Devices” on page 1-198](#). The default device driver implements these primitives based on a simple interface provided by the VisualDSP++ simulator and EZ-KIT Lite systems.

All the I/O requests submitted through the default device driver are channeled through the C function `__primIO`. The assembly label has two underscores, `__primIO`. The source for this function, and all the other library routines, can be found under the base installation for VisualDSP++ in the subdirectory `TS\lib\src\libio_src`.

The `__primIO` function accepts no arguments. Instead, it examines the I/O control block at the label `_PrimIOCB`. Without external intervention by a host environment, the `__primIO` routine simply returns, which indicates failure of the request. Two schemes for host interception of I/O requests are provided.

The first scheme is to modify control flow into and out of the `__primIO` routine. Typically, this would be achieved by a break point mechanism available to a debugger/simulator. Upon entry to `__primIO`, the data for

the request will reside in a control block at the label `_PrimIOCB`. If this scheme is used, the host should arrange to intercept control when it enters the `__primIO` routine, and, after servicing the request, return control to the calling routine.

The second scheme involves communicating with the DSP process through a pair of simple semaphores. This scheme is most suitable for an externally-hosted development board. Under this scheme, the host system should clear the data word whose label is `__lone_SHARC`; this will cause `__primIO` to assume that a host environment is present and able to communicate with the DSP process.

If `__primIO` sees that `__lone_SHARC` is cleared, then upon entry (for example, when an I/O request is made) it will set a non-zero value into the word labeled `__Godot`. The `__primIO` routine will then busy-wait until this word is reset to zero by the host. The non-zero value of `__Godot` raised by `__primIO` is the address of the I/O control block.

Data Packing For Primitive I/O

The DSP implementation is based on a word-addressable machine, with each word comprising a fixed number of 8-bit bytes. All `READ` and `WRITE` requests specify a move of some number of 8-bit bytes, that is, the relevant fields count 8-bit bytes, not words. Packing is always little endian, the first byte of a file read or written is the low-order byte of the first word transferred.

Data packing is set to four bytes per word for the TigerSHARC processors. Data packing can be changed on the DSP side to accommodate other DSP architectures by modifying the constant `BITS_PER_WORD`, defined in `_wordsize.h`. (For example, a processor with 16-bit addressable words would change this value to 16).

Note that the file name provided in an `OPEN` request uses the processor's "native" string format, normally one byte per word. Data packing applies only to `READ` and `WRITE` requests.

Data Structure for Primitive I/O

The I/O control block is declared in `_primio.h`, as follows.

```
typedef struct
{
    enum
    {
        PRIM_OPEN = 100,
        PRIM_READ,
        PRIM_WRITE,
        PRIM_CLOSE,
        PRIM_SEEK,
        PRIM_REMOVE,
        PRIM_RENAME
    } op;
    int    fileID;
    int    flags;
    unsigned char *buf; /* data buffer, or file name */
    int    nDesired; /* number of characters to read */
                    /* or write */
    int    nCompleted; /* number of characters actually */
                    /* read or written */
    void *more; /* for future use */
}
PrimIOCB_T;
```

The first field, `op`, identifies which of the seven currently-supported operations is being requested.

The file ID for an open file is a non-negative integer assigned by the debugger or other “host” mechanism. The `fileID` values 0, 1, and 2 are pre-assigned to `stdin`, `stdout`, and `stderr`, respectively. No open request is required for these file IDs. This field is not used for either a `REMOVE` or `RENAME` operation.

The `flags` field is a bit field containing other information for special requests. Meaningful bit values for an `OPEN` operation are:

C and C++ Run-Time Libraries Guide

```
M_OPENR = 0x0001    /* open for reading          */
M_OPENW = 0x0002    /* open for writing        */
M_OPENA = 0x0004    /* open for append        */
M_TRUNCATE = 0x0008 /* truncate to zero length if file exists */
M_CREATE = 0x0010   /* create the file if necessary */
M_BINARY = 0x0020   /* binary file (vs. text file) */
```

For a `READ` operation, the low-order four bits of the `flag` value contain the number of bytes packed into each word of the read buffer, and the rest of the value is reserved for future use.

For a `WRITE` operation, the low-order four bits of the `flag` value contain the number of bytes packed into each word of the write buffer, and the rest of the value form a bit field, for which only the following bit is currently defined:

```
M_ALIGN_BUFFER = 0x10
```

If this bit is set for a `WRITE` request, the `WRITE` operation is expected to be aligned on a DSP word boundary by writing padding NULs to the file before the buffer contents are transferred.

For a `SEEK` request, the `flags` field indicates the seek mode (whence) as follows:

```
enum
{
    M_SEEK_SET = 0x0001,    /* seek origin is the start of
                             the file */
    M_SEEK_CUR = 0x0002,    /* seek origin is the current
                             position within the file */
    M_SEEK_END = 0x0004,    /* seek origin is the end of
                             the file */
};
```

The `flags` field is unused for a `CLOSE`, `RENAME`, or `REMOVE` request.

The `buf` field contains a pointer to the file name for an `OPEN` or `REMOVE` request, or a pointer to the data buffer for a `READ` or `WRITE` request.

The `nDesired` field is set to the number of bytes that should be transferred for a `READ` or `WRITE` request. This field is also used by a `RENAME` request, and is set to a pointer to the new file name.

For a `SEEK` request, the `nDesired` field contains the offset at which the file should be positioned, relative to the origin specified by the `flags` field. (On architectures that only support 16-bit `ints`, the 32-bit offset at which the file should be positioned is stored in the combined fields `[buf, nDesired]`).

The `nCompleted` field is set by `__primIO` to the number of bytes actually transferred by a `READ` or `WRITE` operation. For a `SEEK` operation, `__primIO` will set this field to the new value of the file pointer. (On architectures that only support 16-bit `ints`, `__primIO` will set the new value of the file pointer in the combined fields `[nCompleted, more]`).

The `more` field is reserved for future use, and currently is always set to `NULL` before calling `__primIO`.

stdlib.h

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

This header also provides prototypes for a number of additional integer math functions provided by Analog Devices, such as `avg`, `max`, `clip`; also `count_ones` and `addbitrev`.



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

C and C++ Run-Time Libraries Guide

A number of functions (including `abs`, `avg`, `max`, `min`, `clip`, `count_ones`, and `addbitrev`) are implemented via intrinsics (provided the header file has been `#include'd`) which map to single machine instructions.



If the header is not included, the library implementation is used instead, at a considerable loss in efficiency.

string.h

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

time.h

The `time.h` header file contains standard definitions for time-handling functions. TigerSHARC processors do not support the underlying functionality.

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 `ccts` compiler are:

- [“complex.h — Basic Complex Arithmetic Functions” on page 3-23](#)
- [“filter.h — DSP Filters and Transformations” on page 3-24](#)
- [“matrix.h — Matrix Functions” on page 3-26](#)
- [“stats.h — Statistical Functions” on page 3-27](#)
- [“vector.h — Vector Functions” on page 3-28](#)
- [“window.h — Window Generators” on page 3-29](#)

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_long_double`. The complex functions are listed with their 32 bits float interface (for example, `float` in C) only. The old style K&R form of representing the parameters is used in this section, for convenience purposes only.



Refer to [Table 3-7](#) for a list of complex functions which are described in detail in “[Run-Time Library Reference](#)” on page 3-41.

The following structures are used to represent complex numbers in rectangular coordinates:

```
typedef struct
{
    float re;
    float im;
} complex_float;

typedef struct
{
    double re;
    double im;
} complex_double;

typedef struct
{
    long double re;
    long double im;
} complex_long_double;
```

filter.h — DSP Filters and Transformations

The `filter.h` header file contains filters used in signal processing. It also includes the A-law and μ -law companders that are used by voice-band compression and expansion applications. This header file also contains functions that perform key signal processing transformations, including Fast Fourier Transforms (FFT) and convolution.

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, and the library will use radix-2 or radix-4 implementations as appropriate.

The library also provides two optimized complex and real FFT functions that have been implemented using a fast radix-2 algorithm; however these functions, `cfftf` and `rcfftf`, have certain requirements that may not be appropriate for some applications.



The functions defined in the `filter.h` header file are listed in [Table 3-8](#) and are described in detail in [“Run-Time Library Reference” on page 3-41](#).

Library functions are provided to initialize a twiddle table for an FFT function. A twiddle table is normally calculated once during program initialization and can be used to 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.

The functions described by this header make certain assumptions about their arguments, in order to achieve high efficiency:

- The FFT routines require that the input, output, and temporary arrays be quad-word aligned, and the twiddle table be long-word aligned.

- The filter routines require that the input array (and coefficients) for FIR be quad-word aligned.
- The A-law and μ -law companders require that the input and output arrays be quad-word aligned.



Failure to observe these constraints will result in incorrect operation.

libsim.h — Simulator Services

The `libsim.h` header file defines services that are provided by the VisualDSP++ debugging environment. The header file contains the function `__emuc1k`, which returns the current simulator cycle count. The header file also contains several utility print routines that may be useful for assembly language development since they are much easier to call than `printf`. [Table 3-4](#) lists the utility print routines and provides a brief description of each.



The list of routines contains names suitable for calling from a C program. If you call them from an assembly program, add an additional leading underscore to the name.



These functions conform to the C calling conventions, as described in [“C/C++ Run-Time Model and Environment”](#) on page 1-185.

Table 3-4. libsim Print Routines

Function	Description
<code>__print_int</code>	prints 32-bit int
<code>__print_uint</code>	prints unsigned 32-bit int
<code>__print_float</code>	prints 32-bit float
<code>__print_double</code>	prints 64-bit long double float
<code>__print_longlong</code>	prints 64-bit signed int
<code>__print_ulonglong</code>	prints 64-bit unsigned int

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 3-23 for definitions of the complex types.



Refer to [Table 3-10](#) for a list of matrix functions which are described in detail in “[Run-Time Library Reference](#)” on page 3-41.

The matrix functions are listed with their 32-bit float interface (for example, `float` in C) only. The old style K&R form of representing the parameters is used in this section for convenience purposes only. The matrix functions are each provided in three forms—for the three floating-point types. For brevity, only the `float` form is listed in the detailed descriptions.

For example, under `cvecsadd`, the following list is provided in the **Synopsis**:

```
cvecsaddf(float a[], float b, float c[], int n)
```

But, the library also provides:

```
cvecsadd(double a[], double b, double c[], int n)
cvecsaddl(long double a[], long double b, long double c[], int n)
```

The functions described by this header make certain assumptions about their arguments, in order to achieve high efficiency:

- Input array arguments are constant; for example, 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.

- Input array arguments are quad-word aligned, and output array arguments are at least double-word aligned. The compiler provides this alignment for all top-level arrays; you must not pass an argument which points at an arbitrary, non-aligned location in an array.



Failure to observe these constraints will result in incorrect operation.

stats.h — Statistical Functions

The `stats.h` header file contains statistical functions, such as `autocoh` and `crosscoh`. The statistics routines make the following assumptions about their argument:

- Input array arguments are constant; for example, their contents do not change during the course of the routine. In particular, this means the input arguments do not overlap with any output arguments.
- Input array arguments are quad-word aligned, and output array arguments are at least double-word aligned. The compiler provides this alignment for all top-level arrays; you must not pass an argument that points at an arbitrary, non-aligned location in an array.



Failure to observe these constraints will result in incorrect operation.



Refer to [Table 3-12](#) for a list of statistical functions that are described in detail in “[Run-Time Library Reference](#)” on [page 3-41](#).

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 3-23](#) for the definitions of the complex types.

 Refer to [Table 3-15](#) for a list of vector functions which are described in detail in [“Run-Time Library Reference” on page 3-41](#).

The vector functions are listed with their 32-bit float interface (for example, `float` in C) only. The old style K&R form of representing the parameters is used in this section for convenience purposes only.

The vector functions are each provided in three forms—for the three floating-point types. For brevity, only the `float` form is listed in the detailed descriptions. For example, under `cvecsadd`, the following list is provided in the **Synopsis**.

```
cvecsaddf(float a[], float b, float c[], int n)
```

but the library also provides

```
cvecsadd(double a[], double b, double c[], int n)
cvecsadd(long double a[], long double b, long double c[], int n)
```

The functions described by this header make certain assumptions about their arguments, in order to achieve high efficiency:

- Input array arguments are constant; that is, their contents will not change during the course of the routine. In particular, this means the input arguments do not overlap with any output argument.
- Input array arguments are quad-word aligned, and output array arguments are at least double-word aligned. The compiler provides this alignment for all top-level arrays; do not pass an argument which points at an arbitrary, non-aligned location in an array.

 Failure to observe these constraints results in incorrect operation.

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 3-5 on page 3-29](#).

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 3-5. Window Generator Functions

Description	Prototype
generate bartlett window	<code>void gen_bartlett (float w[], int a, int N)</code>
generate blackman window	<code>void gen_blackman (float w[], int a, int N)</code>
generate gaussian window	<code>void gen_gaussian (float w[], float alpha, int a, int N)</code>
generate hamming window	<code>void gen_hamming (float w[], int a, int N)</code>
generate hanning window	<code>void gen_hanning (float w[], int a, int N)</code>
generate harris window	<code>void gen_harris (float w[], int a, int N)</code>
generate kaiser window	<code>void gen_kaiser (float w[], float beta, int a, int N)</code>
generate rectangular window	<code>void gen_rectangular (float w[], int a, int N)</code>
generate triangle window	<code>void gen_triangle (float w[], int a, int N)</code>
generate von hann window	<code>void gen_vonhann (float w[], int a, int N)</code>

Abridged C++ Library Support

When in C++ mode, the `ccts` 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 3-30](#)
- [“C++ Header Files for C Library Facilities” on page 3-33](#)
- [“Embedded Standard Template Library Header Files” on page 3-34](#)

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` header defines the template class `complex` and a set of associated arithmetic operators. Predefined types include `complex_float` and `complex_long_double`.

This implementation does not support the full set of complex operations as specified by the C++ standard. In particular, it does not support either the transcendental functions or the I/O operators `<<` and `>>`.

The `complex` header and the C library header file `complex.h` refer to two different and incompatible implementations of the `complex` data type.

exception

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

fract

The `fract` header defines the `fract` data type, which supports fractional arithmetic, assignment, and type-conversion operations. The header file is fully described under [“C++ Fractional Type Support” on page 1-168](#).

fstream

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

iomanip

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

ios

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



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

iosfwd

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

iostream

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

istream

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



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

new

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

ostream

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

sstream

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

stdexcept

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

streambuf

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



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

string

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

C++ Header Files for C Library Facilities

For each C standard library header file 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 header 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 3-6](#) is the same as including the equivalent C standard library header file.

[Table 3-6](#) lists the C++ header files that provide access to the C library facilities.

Table 3-6. C++ Header Files for C Library Facilities

Header	Description
<code>cassert</code>	Enforces assertions during function executions
<code>cctype</code>	Classifies characters
<code>cerrno</code>	Tests error codes reported by library functions
<code>cfloat</code>	Tests floating-point type properties
<code>climits</code>	Tests integer type properties
<code>locale</code>	Adapts to different cultural conventions
<code>cmath</code>	Provides common mathematical operations

C and C++ Run-Time Libraries Guide

Table 3-6. C++ Header Files for C Library Facilities (Cont'd)

Header	Description
<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 `ccts` compiler in C++ mode.

The embedded standard template library headers are listed below.

algorithm

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

deque

The `deque` header defines a deque template container.

functional

The `functional` header defines numerous function objects.

hash_map

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

hash_set

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

iterator

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

list

The `list` header defines a list template container.

map

The `map` header defines two map template containers.

memory

The `memory` header defines facilities for managing memory.

numeric

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

queue

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

set

The `set` header defines two set template containers.

stack

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

C and C++ Run-Time Libraries Guide

utility

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

vector

The `vector` header defines a vector template container.

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

fstream.h

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

omanip.h

The `omanip.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 documented library functions by the header file in which they are located, whereas “[Run-Time Library Reference](#)”, starting [on page 3-41](#), presents the functions in alphabetic order.

Table 3-7. Library Functions in the `complex.h` Header File

arg	cabs	cadd
cartesian	cdiv	cexp
cmlt	conj	csub
norm	polar	

Table 3-8. Library Functions in the `filter.h` Header File

a_compress	a_expand	cfft
cfft2d	cfft	convolve
conv2d	fir	fir_decima
fir_interp	ifft	ifft2d
iir	mu_compress	mu_expand
rfft	rfft2d	rfftf
twidfft	twidfftf	

Table 3-9. Library Functions in the `math.h` Header File

acos	alog	alog10
asin	atan	atan2
ceil	cos	cosh
cot	exp	fabs
favg	fclip	floor

Documented Library Functions

Table 3-9. Library Functions in the `math.h` Header File (Cont'd)

fmax	fmin	fmod
frexp	log	log10
modf	pow	rsqrt
sign	sin	sinh
sqrt	tan	tanh

Table 3-10. Library Functions in the `matrix.h` Header File

cmatmadd	cmatmmlt	cmatmsub
cmatsadd	cmatsmlt	cmatssub
matinv	matmadd	matmmlt
matmsub	matsadd	matsmlt
matssub	transpm	

Table 3-11. Library Functions in the `signal.h` Header File

interrupt , interruptf , interrupts	raise	signal , signalf , signals
---	-----------------------	--

Table 3-12. Library Functions in the `stats.h` Header File

autocoh	autocorr	crosscoh
crosscorr	histogram	mean
rms	var	zero_cross

Table 3-13. Supported Library Functions in `stdio.h` Header File

clearerr	fclose	feof
ferror	fflush	fgetc
fgetpos	fgets	fprintf

Table 3-13. Supported Library Functions in `stdio.h` Header File (Cont'd)

<code>fputc</code>	<code>fputs</code>	<code>fopen</code>
<code>fread</code>	<code>freopen</code>	<code>fscanf</code>
<code>fseek</code>	<code>fsetpos</code>	<code>ftell</code>
<code>fwrite</code>	<code>getc</code>	<code>getchar</code>
<code>gets</code>	<code>perror</code>	<code>printf</code>
<code>putc</code>	<code>putchar</code>	<code>puts</code>
<code>remove</code>	<code>rename</code>	<code>rewind</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 3-14. Library Functions in `stdlib.h` Header File

<code>abs</code>	<code>addbitrev</code>	<code>atof</code>
<code>atoi</code>	<code>atol</code>	<code>atold</code>
<code>atoll</code>	<code>avg</code>	<code>bsearch</code>
<code>clip</code>	<code>count_ones</code>	<code>div</code>
<code>max</code>	<code>min</code>	<code>qsort</code>
<code>rand</code>	<code>srand</code>	<code>strtod</code>
<code>strtof</code>	<code>strtoi</code>	<code>strtol</code>
<code>strtold</code>	<code>strtoll</code>	<code>strtoul</code>
<code>strtoull</code>		

Table 3-15. Library Functions in `vector.h` Header File

<code>cvecdot</code>	<code>cvecsadd</code>	<code>cvecsmult</code>
<code>cvecssub</code>	<code>cvecvadd</code>	<code>cvecvsub</code>
<code>cvecvmlt</code>	<code>vecdot</code>	<code>vecsadd</code>

Documented Library Functions

Table 3-15. Library Functions in `vector.h` Header File (Cont'd)

vecsmult	vecssub	vecvadd
vecvmlt	vecvsub	

Table 3-16. Library Functions in `window.h` Header File

gen_bartlett	gen_blackman	gen_gaussian
gen_hamming	gen_hanning	gen_harris
gen_kaiser	gen_rectangular	gen_triangle
gen_vonhann		

Run-Time Library Reference

The C run-time library is a collection of functions that you can call from your C/C++ programs. This section lists the functions in alphabetical order.



The information that follows applies to all of the functions in the library that are described in this reference.

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

Algorithm—High level mathematical representation of the function

Domain—Range of values supported by the function

Notes—Miscellaneous information

a_compress

A-law compression

Synopsis

```
#include <filter.h>
void a_compress(in, out, n)
const in[];      /* Input array */
int 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`. The function has been optimized and requires that both the input and output vectors are quad-word aligned.

Algorithm

$C(k) = \text{a-law compression of } A(k)$

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 int in[];      /* Input array */
int 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`. The function has been optimized and requires that both the input and output vectors are quad-word aligned.

Algorithm

$C(k) = \text{a-law expansion of } A(k)$

for $k=0$ to $n-1$

Domain

Content of input array: 0 to 255

abs

absolute value

Synopsis

```
#include <stdlib.h>
int abs(int x);
long int labs (long int x);
long long int llabs (long long int x);
```

Description

The abs functions return the absolute value of their argument. These functions are built-in functions that cause the compiler to emit an inline instruction to perform the required operation at the point the function is called.

Use “[fabs](#)” on page 3-112 to calculate the absolute value of a floating-point number.

Algorithm

Return $|x|$

Domain

Full range for given type.

acos

arc cosine

Synopsis

```
#include <math.h>
double acos(double x);
float acosf (float x);
long double acosd (long double 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 $[0, \pi]$.

Algorithm

return = $\cos^{-1}(x)$

Domain

$x = [-1.0 \dots 1.0]$

addbitrev

bit-reversed adder

Synopsis

```
#include <stdlib.h>
int addbitrev (int a,int b);
```

Description

The `addbitrev` function adds the two arguments using the bit-reversed adder. This is binary addition in which the carries are propagated to the right, rather than to the left. Therefore,

`addbitrev(0x50,0x40)` yields `0x30`

This is useful in algorithms, such as the FFT and interleaver.

The function is a built-in function that causes the compiler to emit an inline instruction to perform the required operation at the point that the function is called

Algorithm

See “Description”.

alog

anti-log

Synopsis

```
#include <math.h>

float alogf (float x);
double alog (double x);
long double alogd (long double x);
```

Description

The alog functions calculate the natural (base e) anti-log of their argument. An anti-log function performs the reverse of a log function and is therefore equivalent to exponentiation.

The value `HUGE_VAL` will be returned if the argument `x` is greater than the function's domain. For input values less than the domain, the functions will return 0.0.

Algorithm

$$c = e^x$$

Domain

$x = [-87.33, 88.72]$	for <code>alogf()</code>
$x = [-708.39, 709.78]$	for <code>alogd()</code>

Example

```
#include <math.h>

double y;
y = alog(1.0);           /* y = 2.71828... */
```

Run-Time Library Reference

See Also

[alog10](#), [exp](#), [log](#), [pow](#)

alog10

base 10 anti-log

Synopsis

```
#include <math.h>

float alog10f (float x);
double alog10 (double x);
long double alog10d (long double x);
```

Description

The `alog10` functions calculate the base 10 anti-log of their argument. An anti-log function performs the reverse of a `log` function and is therefore equivalent to exponentiation. Therefore, `alog10(x)` is equivalent to `exp(x * log(10.0))`.

The value `HUGE_VAL` will be returned if the argument `x` is greater than the function's domain. For input values less than the domain, the functions will return 0.0.

Algorithm

$$c = e^{(x * \log(10.0))}$$

Domain

`x` = [-37.92 , 38.53] for `alog10f()`

`x` = [-307.65 , 308.25] for `alog10d()`

Example

```
#include <math.h>
```

Run-Time Library Reference

```
double y;  
y = alog10(1.0);           /* y = 10.0 */
```

See Also

[alog](#), [exp](#), [log10](#), [pow](#)

arg

get phase of a complex number

Synopsis

```
#include <complex.h>

float argf (complex_float a);
double arg (complex_double a);
long double argd (complex_long_double a);
```

Description

These functions compute the phase associated with a Cartesian number represented by the complex argument *a*, and return the result.

Algorithm

$$c = \operatorname{atan}\left(\frac{\operatorname{Im}(a)}{\operatorname{Re}(a)}\right)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$ for `argf()`

-1.7×10^{308} to $+1.7 \times 10^{308}$ for `argd()`

asin

arc sine

Synopsis

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

Description

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

Algorithm

$return = \sin^{-1}(x)$

Domain

$x = [-1.0 \dots 1.0]$

atan

arc tangent

Synopsis

```
#include <math.h>
double atan(double x);
float atanf (float x);
long double atand (long double x);
```

Description

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

Algorithm

return = $\tan^{-1}(x)$

Domain

$x = [-3.4 \times 10^{38} \text{ to } +3.4 \times 10^{38}]$ for `atanf()`

$x = [-1.7 \times 10^{308} \text{ to } 1.7 \times 10^{308}]$ for `atand()`

atan2

arc tangent of quotient

Synopsis

```
#include <math.h>
double atan2 (double x, double y);
float atan2f (float x, float y);
long double atan2d (long double x, long double y);
```

Description

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

Algorithm

$$return = \tan^{-1}\left(\frac{y}{x}\right)$$

Domain

$x, y = [-3.4 \times 10^{38} \text{ to } +3.4 \times 10^{38}]$ for `atan2()`

$x, y = [-1.7 \times 10^{308} \text{ to } 1.7 \times 10^{308}]$ for `atan2d()`

NOT $y = 0$ and $x \neq 0$

Error Conditions

The `atan2` function returns a zero if $y = 0$ and $x \neq 0$.

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 where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `nptr` argument may represent either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (`+`) or minus (`-`); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (`.`).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

Run-Time Library Reference

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix 0x or 0X . This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter p or P, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens [hexdigs] [.hexdigs].

The first character that does not fit either form of number will stop the scan.

Domain

The number should fit within the dynamic range of a `double`. The function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `HUGE_VAL` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Notes

The function reference `atof (pdata)` is functionally equivalent to:

```
strtod (pdata, (char *) NULL);
```

and therefore, if the function returns zero, it is not possible to determine whether the character string contained a (valid) representation of 0.0 or some invalid numerical string.

atoi

convert string to integer

Synopsis

```
#include <stdlib.h>
int atoi (const char *string)
```

Description

The `atoi` function converts a character string to an integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The string argument is as follows:

```
[whitespace] [sign] [base] digits
```

where *whitespace* can consist of spaces and/or tab characters, *sign* is either plus (+) or minus (-), *base* is 0 for octal and 0x for hexadecimal, and *digits* are one or more decimal digits (or letters from a to f for hexadecimal numbers).

The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above.

Domain

-2,147,483,648 to 2,147,483,647

atol

convert string to long integer

Synopsis

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

Description

The `atol` function converts a character string to a long integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The string argument is as follows:

```
[whitespace][sign][base]digits
```

where *whitespace* can consist of spaces and/or tab characters, *sign* is either plus (+) or minus (-), *base* is 0 for octal and 0x for hexadecimal, and *digits* are one or more decimal digits (or letters from a to f for hexadecimal numbers).

The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above.

Domain

-2,147,483,648 to 2,147,483,647

atold

convert string to long double

Synopsis

```
#include <stdlib.h>
long double atold (const char *string)
```

Description

The `atold` function converts a character string to a double-precision floating-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `nptr` argument may represent either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (`+`) or minus (`-`); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (`.`).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

Run-Time Library Reference

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix 0x or 0X . This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter p or P, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens [hexdigs] [.hexdigs].

The first character that does not fit either form of number will stop the scan.

Domain

The number should fit within the dynamic range of a long double. The function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `LDBL_MAX` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Notes

The function reference `atold (pdata)` is functionally equivalent to:

```
strtold (pdata, (char *) NULL);
```

and therefore if the function returns zero, it is not possible to determine whether the character string contained a (valid) representation of 0.0 or some invalid numerical string.

atoll

convert string to long long integer

Synopsis

```
#include <stdlib.h>
long long atoll (const char *string)
```

Description

The `atoll` function converts a character string to a long long integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The string argument is as follows:

```
[whitespace] [sign] [base] digits
```

where *whitespace* can consist of spaces and/or tab characters, *sign* is either plus (+) or minus (-), *base* is 0 for octal and 0x for hexadecimal, and *digits* are one or more decimal digits (or letters from a to f for hexadecimal numbers).

The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above.

Domain

-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

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 */
```

Description

The `autocoh` function computes the autocohereence of the input elements contained within input vector `a`, and stores the result to output vector `c`.

Algorithm

$$c_k = \frac{1}{n} * \left(\sum_{j=0}^{n-k-1} (a_j - \bar{a}) * (a_{j+k} - \bar{a}) \right)$$

where $k=\{0,1,...,m-1\}$ and $\bar{a}$ is the mean value of input vector `a`.

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

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         */

```

Description

The `autocorr` 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 $+3.4 \times 10^{38}$

avg

mean of two values

Synopsis

```
#include <stdlib.h>

int avg(int a, int b);
long int lavg (long int a, long int b);
long long int llavg (long long int a, long long int b);
```

Description

The avg functions add the two arguments and divide the result by two. These functions are built-in functions that cause the compiler to emit an inline instruction to perform the required operation at the point the function is called.

Algorithm

$$(a + b)/2$$

Domain

Full range.

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.

Algorithm

Call the compare function with the key and the current node (initially the one in the middle of the array). If the compare returns 0, then return the current node; if the compare returns -1, apply the search recursively on

Run-Time Library Reference

the portion of the array to the left of the current position; if the compare returns 1, apply the search recursively on the portion of the array to the right of the current location.

Domain

N/A

cabs

complex absolute value

Synopsis

```
#include <complex.h>
float cabsf(a)
complex_float a;      /* Complex input */
```

Description

The `cabs` 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( )`

cadd

complex addition

Synopsis

```
#include <complex.h>
complex_float caddf(a,b)
complex_float a;          /* Complex input a */
complex_float b;          /* Complex input b */
```

Description

The `cadd` function adds two complex values a and b , and returns the result.

Algorithm

$$\begin{aligned}\operatorname{Re}(c) &= \operatorname{Re}(a) + \operatorname{Re}(b) \\ \operatorname{Im}(c) &= \operatorname{Im}(a) + \operatorname{Im}(b)\end{aligned}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cartesian

convert Cartesian to polar notation

Synopsis

```
#include <complex.h>

float cartesianf (complex_float a, float *phase);
double cartesian (complex_double a, double *phase);
long_double cartesiand (complex_long_double a,
                        long double *phase);
```

Description

These functions transform a complex number from Cartesian notation to polar notation. The Cartesian number is represented by the argument `a` that the function converts into a corresponding magnitude, which is returned as the function's result, and a phase that is returned via the second argument `phase`.

Algorithm

```
magnitude = cabs(a)
phase = arg(a)
```

Domain

$[-3.4 \times 10^{38} \text{ to } +3.4 \times 10^{38}]$	for <code>cartesianf()</code>
$[-1.7 \times 10^{308} \text{ to } 1.7 \times 10^{308}]$	for <code>cartesiand()</code>

Example

```
#include <complex.h>

complex_float point = {-2.0 , 0.0};
float phase;
```

Run-Time Library Reference

```
float mag;  
mag = cartesianf (point,&phase);      /* mag = 2.0, phase =  $\pi$  */
```

cdiv

complex division

Synopsis

```
#include <complex.h>
complex_float cdivf(a,b)
complex_float a;          /* Complex input a */
complex_float b;          /* Complex input b */
```

Description

The `cdiv` function computes the quotient of the division of two complex values 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 $+3.4 \times 10^{38}$

ceil

ceiling

Synopsis

```
#include <math.h>
float ceilf (float x)
double ceil (double x)
long double ceild (long double x)
```

Description

The `ceil` function calculates the next highest whole number that is greater than or equal to the floating-point number `x`. It returns the smallest integral value, that is not less than its input.

Algorithm

return = smallest int $\geq x$

Domain

$x = [-3.4 \times 10^{38} \text{ to } +3.4 \times 10^{38}]$	for <code>ceilf()</code>
$x = [-1.7 \times 10^{308} \text{ to } 1.7 \times 10^{308}]$	for <code>ceild()</code>

cexp

complex exponential

Synopsis

```
#include <complex.h>
complex_float cexpf(a)
float a;          /* Value of input */
```

Description

The `cexp` 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` = [-1,647,095 ... 1,647,095] for `cexpf()`

`a` = [-843,314,850 ... 843,314,850] for `cexpd()`

cfft

N point complex input FFT

Synopsis

```
#include <filter.h>

void cfft(in[], t[], out[], w[], wst, n)
const complex_float in[];      /* Input sequence          */
complex_float t[];             /* Temporary working buffer */
complex_float out[];           /* Output sequence         */
const complex_float w[];       /* Twiddle sequence        */
int wst;                       /* Twiddle factor stride    */
int n;                         /* Number of FFT points    */
```

Description

The `cfft` function transforms the time domain complex input signal sequence to the frequency domain by using the accelerated version of the ‘Discrete Fourier Transformation’ known as a ‘Fast Fourier Transform’ or FFT. The `cfft` function will “decimate in frequency” by the best choice FFT algorithm, radix-4 or mixed radix, depending on the input sequence length.

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. If the input data can be overwritten, then the memory requirements may be reduced by specifying the input array as the output array. Run-time performance of the function will be improved if the input and output arrays are allocated in a different memory block than the twiddle table, `w`.

The twiddle table is passed in the argument `w`, which must contain at least $\frac{3}{4}n$ complex twiddle factors. The function `twidfft` may be used to initialize the array. If the twiddle table contains more factors than needed for a

particular call on `cfft`, then the stride factor has to be set appropriately; otherwise it should be 1. Refer to [“twidfft” on page 3-212](#) for more information.

i The library also contains the `cfftfc` function (see [on page 3-81](#)), which is an optimized implementation of a complex FFT using a fast radix-2 algorithm. The `cfftfc` function however imposes certain memory alignment requirements that may not be appropriate for some applications.

Algorithm

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk}$$

When the sequence length, n , is a power of four, the radix-4 method is used. When the sequence length is a power of two (see “Domain”), the mixed radix method is used. At the first stage of the decimation in frequency, the `cfft` function uses the radix-2 method to generate two sequences with $n/2$ points each. The $n/2$ is now a power of four which creates the condition to employ the faster radix-4 method over these two sequences.

Domain

Input sequence length n must be equal to either a power of two, or a power of four, and at least 16.

Example

```
/* Example to demonstrate how to generate two complex FFTs using
   a single twiddle table */

#include <filter.h>

#define NDATA1 256
```

Run-Time Library Reference

```
#define NDATA2 32

complex_float data1[NDATA1];  /* data for a 256-point FFT */
complex_float data2[NDATA2];  /* data for a 32-point FFT */

complex_float output1[NDATA1];
complex_float output2[NDATA2];

static complex_float twidtab[(3*NDATA1)/4];
complex_float temp[NDATA1];
    /* note that the temporary buffer should be as large as
       the largest FFT generated */

/* Generate a twiddle table for a 256-point FFT */

twidfft (twidtab, NDATA1);
    /* note that a twiddle table is constant for a given number
       of FFT points */

/* Generate a 256-point complex FFT */

cfft (data1, temp, output1, twidtab, 1, NDATA1);
    /* note that the twiddle table stride factor is 1 */

/* Generate a 32-point complex FFT */

cfft (data2, temp, output2, twidtab, (NDATA1/NDATA2), NDATA2);
    /* note that the twiddle table stride factor is 8 */
```

cfft_mag

cfft magnitude

Synopsis

```
#include <filter.h>

void cfft_mag (const complex_float dm input[],
               float dm output[],
               int fftsize);
```

Description

The `cfft_mag` function computes a normalized power spectrum from the output signal generated by a `cfft` or `cfft_f` function. The size of the signal and the size of the power spectrum is `fftsize`.



The Nyquist frequency is located at $(\text{fftsize}/2) + 1$.

Algorithm

$$\text{magnitude}(z) = \frac{\sqrt{\text{Re}(z)^2 + \text{Im}(z)^2}}{\text{fftsize}}$$

Example

```
#include <filter.h>
#define N 64

complex_float fft_input[N];
complex_float fft_output[N];

complex_float temp[N];
complex_float twid[(3*N)/4];

float spectrum[N];
```

Run-Time Library Reference

```
/* Generate a twiddle table for the FFT */  
  
twidfft (twid,N);  
  
    /* Note that a twiddle table is constant for a given number  
      of FFT points */  
  
/* Generate a complex FFT using cfft */  
  
cfft (fft_input, temp, fft_output, twid, 1, N);  
  
/* Generate the power spectrum */  
  
cfft_mag (fft_output, spectrum, N);
```

cfft2d

NxN point 2-D complex input FFT

Synopsis

```

#include <filter.h>
void cfft2d(*in, *t, *out, w[], wst, n)
const complex_float *in; /* Pointer to input matrix a[n][n] */
complex_float *t;        /* Pointer to working buffer t[n][n] */
complex_float *out;      /* Pointer to matrix c[n][n] */
const complex_float w[]; /* Twiddle sequence */
int wst;                  /* Twiddle factor stride */
int n;                    /* Number of FFT points */

```

Description

The `cfft2d` function computes the two-dimensional Fast Fourier Transform of the complex input matrix `a[n][n]`, and stores the result to the complex matrix `c[n][n]`.

If the input data can be overwritten, optimum memory usage is achieved by setting the output pointer to the input array.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `cfft2d`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 3-212.](#)

Run-Time Library Reference

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 equal to either a power of two, or a power of four, and at least 16.

cfft

fast N point complex input FFT

Synopsis

```
#include <filter.h>
void cfft(in[], out[], twid[], wst, n)
const complex_float in[];      /* Input sequence      */
complex_float out[];           /* Output sequence     */
const complex_float twid[];     /* Twiddle sequence    */
int wst;                       /* Twiddle factor stride */
int n;                         /* Number of FFT points */
```

Description

The `cfft` function transforms the time domain complex input signal sequence to the frequency domain by using the accelerated version of the ‘Discrete Fourier Transform’ known as a ‘Fast Fourier Transform’ or FFT. It will “decimate in frequency” using an optimized radix-2 algorithm.

The size of the input array `in` and the output array `out` is `n`, where `n` represents the number of points in the FFT. The `cfft` function has been designed for optimum performance and requires that the input array `in` be aligned on an address boundary that is a multiple of twice the FFT size. For certain applications, this alignment constraint may not be appropriate; in such cases, the application should call the `cfft` function instead with no loss of facility (apart from performance).

The twiddle table is passed in the argument `twid`, which must contain at least $n/2$ complex twiddle factors. The function `twidfft` may be used to initialize the array. If the twiddle table contains more factors than required for a particular FFT size, then the stride factor `wst` has to be set appropriately; otherwise, it should be set to 1. [For more information, see “twidfft” on page 3-214.](#)



The twiddle tables used by the functions `cfft` and `cfft_f` are not compatible. The `cfft` function (see [on page 3-74](#)) uses a twiddle table that contains $\frac{3}{4}n$ factors in which the imaginary coefficients are positive sine values, while the `cfft_f` function uses a twiddle table with $\frac{1}{2}n$ factors in which the imaginary coefficients are negative sine values.

It is recommended that the twiddle table and the output array are allocated in separate memory blocks—otherwise, the performance of the function will degrade.

Algorithm

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk}$$

Domain

The number of points in the FFT must be a power of 2 and must be at least 32.

clip

clip

Synopsis

```
#include <stdlib.h>

int clip (int parm1, int parm2);
long int lclip (long int parm1, long int parm2);
long long int llclip (long long int parm1, long long int parm2);
```

Description

The clip functions return their first argument if it is less than the absolute value of the second argument, otherwise they return the absolute value of the second argument if the first is positive, or minus the absolute value if the first argument is negative. The clip functions are built-in functions which are implemented with a CLIP instruction.

Algorithm

```
if ( |parm1| < |parm2| )
    return( parm1 )
else
    return( |parm2| * signof(parm1))
```

Domain

Full range.

cmatmadd

complex matrix + matrix addition

Synopsis

```
#include <matrix.h>
void cmatmaddf(a,b,n,m,c)
const complex_float *a;    /* Pointer to input matrix a[][] */
const complex_float *b;    /* Pointer to input matrix b[][] */
int n;                     /* Number of rows in matrix a[][] */
int m;                     /* Number of columns in matrix a[][] */
complex_float *c;          /* Pointer to matrix c[][] */
```

Description

This function computes the addition of input complex matrix $a[][]$ with input complex matrix $b[][]$, and stores the result to output complex matrix $c[][]$. The dimensions of complex matrix $a[][]$ are n and m and the dimensions of complex matrix b are n and m . The resulting output complex matrix $c[][]$ is of dimensions n and m .

Algorithm

$$\text{Re}(c_{i,j}) = \text{Re}(a_{i,j}) + \text{Re}(b_{i,j})$$

$$\text{Im}(c_{i,j}) = \text{Im}(a_{i,j}) + \text{Im}(b_{i,j})$$

where $i=\{0,1,2,\dots,n-1\}$, $j=\{0,1,2,\dots,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cmatmmltf

complex matrix * matrix multiplication

Synopsis

```
#include <matrix.h>
void cmatmmltf(a,n,k,b,m,c)
const complex_float *a; /* Pointer to input matrix a[][] */
int n; /* Number of rows in matrix a[][] */
int k; /* Number of columns in matrix a[][] */
const complex_float *b; /* Pointer to input matrix b[][] */
int m; /* Number of columns in matrix b[][] */
complex_float *c; /* Pointer to matrix c[][] */
```

Description

This function computes the multiplication of input complex matrix $a[][]$ with input complex matrix $b[][]$, and stores the result to output complex matrix $c[][]$. The dimensions of complex matrix $a[][]$ are n and k and the dimensions of complex matrix b are k and m . The resulting output complex matrix $c[][]$ is of dimensions n and m .

Algorithm

$$\text{Re}(c_{i,j}) = \sum_{l=0}^{k-1} (\text{Re}(a_{i,l}) * \text{Re}(b_{l,j}) - \text{Im}(a_{i,l}) * \text{Im}(b_{l,j}))$$

$$\text{Im}(c_{i,j}) = \sum_{l=0}^{k-1} (\text{Re}(a_{i,l}) * \text{Im}(b_{l,j}) + \text{Im}(a_{i,l}) * \text{Re}(b_{l,j}))$$

where $i=\{0,1,2,...,n-1\}$, $j=\{0,1,2,...,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cmatmsub

complex matrix - matrix subtraction

Synopsis

```
#include <matrix.h>
void cmatmsubf(a,b,n,m,c)
const complex_float *a;    /* Pointer to input matrix a[][] */
int n;                     /* Number of rows in matrix a[][] */
int k;                     /* Number of columns in matrix a[][] */
const complex_float *b;    /* Pointer to input matrix b[][] */
int m;                     /* Number of columns in matrix b[][] */
complex_float *c;          /* Pointer to matrix c[][] */
```

Description

This function computes the subtraction of input complex matrix $a[][]$ with input complex matrix $b[][]$, and stores the result to output complex matrix $c[][]$. The dimensions of complex matrix $a[][]$ are n and m and the dimensions of complex matrix b are n and m . The resulting output complex matrix $c[][]$ is of dimensions n and m .

Algorithm

$$\text{Re}(c_{i,j}) = \text{Re}(a_{i,j}) - \text{Re}(b_{i,j})$$

$$\text{Im}(c_{i,j}) = \text{Im}(a_{i,j}) - \text{Im}(b_{i,j})$$

where $i=\{0,1,2,\dots,n-1\}$, $j=\{0,1,2,\dots,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cmatsadd

complex matrix + scalar addition

Synopsis

```

#include <matrix.h>
void cmatsaddf(a,b,n,m,c)
const complex_float *a;  /* Pointer to input matrix a[][] */
const complex_float b;    /* Input scalar b */
int n;                   /* Number of rows in matrix a[][] */
int m;                   /* Number of columns in matrix a[][] */
complex_float *c;        /* Pointer to matrix c[][] */

```

Description

This function adds the complex scalar value b to each element of complex matrix $a[][]$, placing the result in complex matrix $c[][]$. The dimensions of complex matrix $a[][]$ are n and m . The resulting output complex matrix $c[][]$ is of dimensions n and m .

Algorithm

$$\text{Re}(c_{i,j}) = \text{Re}(a_{i,j}) + \text{Re}(b)$$

$$\text{Im}(c_{i,j}) = \text{Im}(a_{i,j}) + \text{Im}(b)$$

where $i=\{0,1,2,\dots,n-1\}$, $j=\{0,1,2,\dots,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cmatmult

complex matrix * scalar multiplication

Synopsis

```
#include <matrix.h>
void cmatmultf(a,b,n,m,c)
const complex_float *a; /* Pointer to input matrix a[][] */
const complex_float b; /* Input scalar b */
int n; /* Number of rows in matrix a[][] */
int m; /* Number of columns in matrix a[][] */
complex_float *c; /* Pointer to matrix c[][] */
```

Description

This function computes the multiplication of input complex matrix $a[][]$ with input complex scalar b , and stores the result to output complex matrix $c[][]$. The dimensions of complex matrix $a[][]$ are n and m . The resulting output complex matrix $c[][]$ is of dimensions n and m .

Algorithm

$$\text{Re}(c_{i,j}) = \text{Re}(a_{i,j}) * \text{Re}(b) - \text{Im}(a_{i,j}) * \text{Im}(b)$$

$$\text{Im}(c_{i,j}) = \text{Re}(a_{i,j}) * \text{Im}(b) + \text{Im}(a_{i,j}) * \text{Re}(b)$$

where $i=\{0,1,2,\dots,n-1\}$, $j=\{0,1,2,\dots,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cmatssub

complex matrix - scalar subtraction

Synopsis

```
#include <matrix.h>
void cmatssubf(a,b,n,m,c)
const complex_float *a;    /* Pointer to input matrix a[][] */
const complex_float b;     /* Input scalar b */
int n;                     /* Number of rows in matrix a[][] */
int m;                     /* Number of columns in matrix a[][] */
complex_float *c;          /* Pointer to matrix c[][] */
```

Description

This function computes the subtraction of input complex matrix $a[][]$ with input complex scalar b , and stores the result to output complex matrix $c[][]$. The dimensions of complex matrix $a[][]$ are n and m . The resulting output complex matrix $c[][]$ is of dimensions n and m .

Algorithm

$$\text{Re}(c_{i,j}) = \text{Re}(a_{i,j}) - \text{Re}(b)$$

$$\text{Im}(c_{i,j}) = \text{Im}(a_{i,j}) - \text{Im}(b)$$

where $i=\{0,1,2,\dots,n-1\}$, $j=\{0,1,2,\dots,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cmlt

complex multiply

Synopsis

```
#include <complex.h>
complex_float cmltf(a,b)
complex_float a;          /* Complex input a */
complex_float b;          /* Complex input b */
```

Description

This function multiplies two complex values 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 $+3.4 \times 10^{38}$

conj

complex conjugate

Synopsis

```
#include <complex.h>
complex_float conjf(a)
complex_float a;      /* Complex input a */
```

Description

This function conjugates the complex input *a*, and returns the result.

Algorithm

$$\text{Re}(c) = \text{Re}(a)$$

$$\text{Im}(c) = -\text{Im}(a)$$

Domain

$$-3.4 \times 10^{38} \text{ to } +3.4 \times 10^{38}$$

convolve

convolution

Synopsis

```
#include <filter.h>
void convolve(vin1, vlen1, vin2, vlen2, vout)
const float vin1[];          /* Pointer to input sequence 1 */
int vlen1;                   /* Length of the input sequence 1 */
const float vin2[];          /* Pointer to input sequence 2 */
int vlen2;                   /* Length of the input sequence 2 */
float vout[];                /* Pointer to output sequence */
```

Description

This function convolves two sequences pointed to by `vin1` and `vin2`. If `vin1` points to the sequence whose length is `vlen1` and `vin2` points to the sequence whose length is `vlen2`, the resulting sequence pointed to by `vout` has the length `vlen1 + vlen2 - 1`.

Algorithm

Convolution between two sequences `cin1` and `cin2` is described as:

$$cout(n) = \sum_{k=0}^{k=vlen2-1} cin1(n+k) \bullet cin2(vlen2-k-1)$$

for $n = \{0, 1, 2, \dots, vlen-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

conv2d

2-D convolution

Synopsis

```

#include <filter.h>
void conv2d(min1, mrow1, mcol1, min2, mrow2, mcol2, mout )
const float *min1;          /* Pointer to input matrix 1 */
int mrow1;                  /* Number of rows in matrix 1 */
int mcol1;                  /* Number of columns in matrix 1 */
const float *min2;          /* Pointer to input matrix 2 */
float *mrow2;               /* Number of rows in matrix 1 */
int mcol2;                  /* Number of columns in matrix 2 */
int *mout;                  /* Pointer to 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`.

$$mout(r, c) = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} min2[k-1-i][k-1-j] \bullet min1[r+i][c+j]$$

for $r=0$ to $nr-k+1$ and for $c=0$ to $nc-k+1$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

copysign

copy the sign

Synopsis

```
#include <math.h>
float copysignf (float parm1, float parm2)
double copysign (double parm1, double parm2)
long double copysignl (long double parm1, long double 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

cos

cosine

Synopsis

```
#include <math.h>
float cosf (float x)
double cos (double x)
long double cosd (long double x)
```

Description

This function calculates the cosine of number x where x is measured in radians. The output is in the range $[-1$ to $1]$. If x is outside of the domain, this function returns 0.0.

Algorithm

return = *cos*(*x*)

Domain

$x = [-1,647,095 \dots 1,647,095]$	for <code>cosf()</code>
$x = [-843,314,850 \dots 843,314,850]$	for <code>cosd()</code>

cosh

hyperbolic cosine

Synopsis

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

Description

This function calculates the hyperbolic cosine of a number x where x is measured in radians. If x is outside the domain, this function returns 3.4×10^{38} for a `float`-type return value and 1.7×10^{308} for a `double`-type return value.

Algorithm

return = cosh(x)

Domain

$x = [-(\ln(3.4 \times 10^{38}) - \ln(2)) \dots (\ln(3.4 \times 10^{38}) - \ln(2))]$ for `coshf()`

$x = [-(\ln(1.7 \times 10^{308}) - \ln(2)) \dots (\ln(1.7 \times 10^{308}) - \ln(2))]$ for `coshd()`

cot

cotangent

Synopsis

```
#include <math.h>
float cotf (float x)
double cot (double x)
long double cotd (long double x)
```

Description

This function calculates the cotangent of number x where x is measured in radians. If x is outside of the domain, this function returns 0.0.

Algorithm

return = cot(x)

Domain

$x = [-6,588,397 \dots 6,588,397]$	for <code>cotf()</code>
$x = [-421,657,424 \dots 421,657,424]$	for <code>cotd()</code>

count_ones

count one bits in word

Synopsis

```
#include <stdlib.h>
int count_ones(int parm)
int lcount_ones(long parm)
int llcount_ones(long long parm)
```

Description

These functions count the number of one bits in the argument `parm`.

Algorithm

$$return = \sum_{j=0}^{j=N-1} bit[j] \text{ of } parm$$

where N is the number of bits in `parm`.

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         */

```

Description

This function computes the cross-coherence of the input elements contained within input vector *a* and input vector *b* and stores the result in the 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

-3.4×10^{38} to $+3.4 \times 10^{38}$

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 */
```

Description

This function computes the cross-correlation of the input elements contained within input vector *a* and input vector *b* and places the result in the 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 $+3.4 \times 10^{38}$

csub

complex subtraction

Synopsis

```
#include <complex.h>
complex_float csubf(a,b)
complex_float a;      /* Complex input a */
complex_float b;      /* Complex input 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 $+3.4 \times 10^{38}$

cvecdot

complex vector dot product

Synopsis

```
#include <vector.h>
complex_float cvecdotf(a,b,n)
const complex_float a[];      /* Input vector a */
const complex_float b[];      /* Input vector b */
int n;                        /* Element count */
```

Description

This function computes the complex dot product of two complex input vectors *a* and *b* and returns the complex result.

Algorithm

The algorithm for a complex dot product is given by:

$$\text{Re}(c_i) = \sum_{l=0}^{n-1} \text{Re}(a_l) * \text{Re}(b_l) - \text{Im}(a_l) * \text{Im}(b_l)$$
$$\text{Im}(c_i) = \sum_{l=0}^{n-1} \text{Re}(a_l) * \text{Im}(b_l) + \text{Im}(a_l) * \text{Re}(b_l)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cvecsadd

complex vector + scalar addition

Synopsis

```
#include <vector.h>
void cvecsaddf(a,b,c,n)
const complex_float a[];      /* Input vector a */
complex_float b;               /* Input scalar b */
complex_float c[];            /* Output vector */
int n;                         /* Element count */
```

Description

This function adds input complex scalar b to each element of input complex vector a and stores the results in the output complex vector c .

Algorithm

$$\text{Re}(c_i) = \text{Re}(a_i) + \text{Re}(b)$$

$$\text{Im}(c_i) = \text{Im}(a_i) + \text{Im}(b)$$

where $i=\{0,1,2,\dots,n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cvecssmlt

complex vector * scalar multiplication

Synopsis

```
#include <vector.h>
void cvecssmlt(f(a,b,c,n)
const complex_float a[];      /* Input vector a */
complex_float b;               /* Input scalar b */
complex_float c[];            /* Output vector */
int n;                         /* Element count */
```

Description

This function multiplies each element of input complex vector *a* by input complex scalar *b* and stores the results in the output complex vector *c*.

Algorithm

$$\text{Re}(c_i) = \text{Re}(a_i) * \text{Re}(b) - \text{Im}(a_i) * \text{Im}(b)$$

$$\text{Im}(c_i) = \text{Re}(a_i) * \text{Im}(b) + \text{Im}(a_i) * \text{Re}(b)$$

where $i=\{0,1,2,\dots,n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cvecssub

complex vector - scalar subtraction

Synopsis

```
#include <vector.h>
void cvecssubf(a,b,c,n)
const complex_float a[];      /* Input vector a */
complex_float b;              /* Input scalar b */
complex_float c[];            /* Output vector */
int n;                        /* Element count */
```

Description

This function subtracts input complex scalar b from each element of input complex vector a and stores the results in the output complex vector c .

Algorithm

$$\text{Re}(c_i) = \text{Re}(a_i) - \text{Re}(b)$$

$$\text{Im}(c_i) = \text{Im}(a_i) - \text{Im}(b)$$

where $i=\{0,1,2,\dots,n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cvecvadd

complex vector + vector addition

Synopsis

```
#include <vector.h>
void cvecvaddf(a,b,c,n)
const complex_float a[];      /* Input vector a */
const complex_float b[];      /* Input vector b */
complex_float c[];            /* Output vector */
int n;                         /* Element count */
```

Description

This function adds two input vectors and stores the results in the output vector c.

Algorithm

$$\begin{aligned}\text{Re}(c_i) &= \text{Re}(a_i) + \text{Re}(b_i) \\ \text{Im}(c_i) &= \text{Im}(a_i) + \text{Im}(b_i) \\ \text{where } i &= \{0, 1, 2, \dots, n-1\}\end{aligned}$$

Domain

$$-3.4 \times 10^{38} \text{ to } +3.4 \times 10^{38}$$

cvecvmlt

complex vector * vector multiplication

Synopsis

```
#include <vector.h>
void cvecvmlt(a,b,c,n)
const complex_float a[];      /* Input vector a */
const complex_float b[];      /* Input vector b */
complex_float c[];            /* Output vector */
int n;                        /* Element count */
```

Description

This function multiplies two input vectors *a* and *b* and stores the results in the results in the output vector *c*.

Algorithm

$$\text{Re}(c_i) = \text{Re}(a_i) * \text{Re}(b_i) - \text{Im}(a_i) * \text{Im}(b_i)$$

$$\text{Im}(c_i) = \text{Re}(a_i) * \text{Im}(b_i) + \text{Im}(a_i) * \text{Re}(b_i)$$

where $i=\{0,1,2,\dots,n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

cvecvsub

complex vector - vector subtraction

Synopsis

```
#include <vector.h>
void cvecvsubf(a,b,c,n)
const complex_float a[];      /* Input vector a */
const complex_float b[];      /* Input vector b */
complex_float c[];            /* Output vector */
int n;                        /* Element count */
```

Description

This function subtracts input vector *b* from input vector *a* and stores the results in the output vector *c*.

Algorithm

$$\text{Re}(c_i) = \text{Re}(a_i) - \text{Re}(b_i)$$

$$\text{Im}(c_i) = \text{Im}(a_i) - \text{Im}(b_i)$$

where $i=\{0,1,2,\dots,n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

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
```

Algorithm

$$Quotient = \left\lfloor \frac{|(numer)|}{|denom|} \right\rfloor \times \text{signof}\left(\frac{(numer)}{(denom)}\right)$$

$$Remainder = numer - (Quotient \times denom)$$

Domain

numerator: -2,147,483,648 to 2,147,483,647
denominator: -2,147,483,648 to -1 and 1 to 2,147,483,647

exp

exponential

Synopsis

```
#include <math.h>
```

```
float expf (float x)
```

```
double exp (double x)
```

```
long double expd (long double x)
```

Description

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

The value `HUGE_VAL` will be returned if the argument x is greater than the function's domain. For input values less than the domain, the functions will return 0.0.

Algorithm

$$return = e^{(x)}$$

Domain

$x = [-87.33, 88.72]$ for `expf()`

$x = [-708.39, 709.78]$ for `expd()`

__emuclk

Get simulator cycle count

Synopsis

```
#include <libsim.h>
int __emuclk(void)
```

Description

This function returns the current value of the simulator cycle count. Note that the function name has two leading underscores.

Algorithm

See “Description”.

fabs

float absolute value

Synopsis

```
#include <math.h>

double fabs(double f);
float fabsf(float f);
long double fabsd (long double f);
```

Description

The `fabs` functions return the absolute value of their argument.

The `fabsf` function is a built-in function which is implemented with a `ABS` instruction; the `fabs` function will be compiled as a built-in function if `double` is the same size as `float`.

Algorithm

return $|x|$

Example

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

y = fabs(-2.3);      /* y = 2.3 */
y = fabs(2.3);       /* y = 2.3 */
```

See Also

[abs](#)

favg

mean of two values

Synopsis

```
#include <math.h>

float favgf (float a, float b);
double favg (double a, double b);
long double favgd (long double a, long double b);
```

Description

The `favg` functions add the two arguments and divide the result by two.

The `favgf` function is a built-in function that causes the compiler to emit an inline instruction to perform the required operation at the point the function is called; the `favg` function will be compiled as a built-in function if `double` is the same size as `float`.

Algorithm

$$(a + b) / 2$$

Domain

Full range.

fclip

clip x by y

Synopsis

```
#include <math.h>

float fclipf (float parm1, float parm2);
double fclip (double parm1, double parm2);
long double fclipd (long double parm1, long double parm2);
```

Description

The `fclip` functions return their first argument if it is less than the absolute value of the second argument; otherwise, they return the absolute value of the second argument if the first is positive, or minus the absolute value if the first argument is negative.

The `fclipf` function is a built-in function which is implemented with a CLIP instruction; the `fclip` function will be compiled as a built-in function if `double` is the same size as `float`.

Algorithm

```
if ( |parm1| < |parm2| )
    return (parm1)
else
    return (|parm2| * signof(parm1))
```

Domain

Full range.

fir

finite impulse response filter

Synopsis

```
#include <filter.h>
void fir(x,y,n,s)
const float x[];      /* Input sample vector x          */
float y[];             /* Output sample vector y          */
int n;                /* Number of input samples         */
fir_state *s;         /* Pointer to filter state structure */
```

The FIR filter function uses the following structure to maintain the state of the filter:

```
typedef struct
{
    float *h;          /* Filter coefficients          */
    float *d;          /* Start of delay line         */
    float *p;          /* Read/Write pointer          */
    int k;              /* Number of coefficients      */
    int l;              /* Interpolation/decimation index */
} fir_state;
```

Description

The `fir` function implements a finite impulse response (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`, in the `filter.h` header file, is available to initialize the structure and is defined as:

Run-Time Library Reference

```
#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 `float` 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`. This field is normally set to an interpolation/decimation index before calling either the `fir_interp` or `fir_decima` functions.

The `fir` function assumes that the input data and the vector containing the filter coefficients are aligned on a quad-word boundary. For optimal performance, the delay line and the output buffer should not be in the same memory block as the filter coefficients.

Algorithm

$$y(k) = \sum_{i=0}^{p-1} h(i) * x(k - i) \text{ for } k = 0, 1, \dots, n$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

fir_decima

FIR decimation filter

Synopsis

```

#include <filter.h>
void fir_decima(x,y,ng,s)
const float x[];          /* Input sample vector x          */
float y[];                 /* Output sample vector y        */
int ng;                   /* Number of samples to generate */
fir_state *s;             /* Pointer to filter state structure */

```

The FIR filter function uses the following structure to maintain the state of the filter:

```

typedef struct
{
    float *h;              /* Filter coefficients          */
    float *d;              /* Start of delay line         */
    float *p;              /* Read/Write pointer          */
    int k;                 /* Number of coefficients      */
    int l;                 /* Interpolation/decimation index */
} fir_state;

```

Description

The `fir_decima` 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 size of the output vector is specified by the argument `ng`, and the number of input samples should be `ng*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:

Run-Time Library Reference

```
#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 `float` 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 `fir_decima` function requires that the filter coefficients are aligned on a quad-word boundary, and that the coefficients are stored in reverse order, thus `s->h[0]` contains the last coefficient.

Algorithm

$$y(k) = \sum_{i=0}^{p-1} x(k * l - i) * h(i)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

fir_interp

FIR interpolation filter

Synopsis

```

#include <filter.h>
void fir_interp(x,y,n,s)
const float x[];          /* Input sample vector x          */
float y[];                 /* Output sample vector y       */
int n;                    /* Number of input samples      */
fir_state *s;             /* Pointer to filter state structure */

```

The FIR filter function uses the following structure to maintain the state of the filter:

```

typedef struct
{
    float *h;              /* Filter coefficients          */
    float *d;              /* Start of delay line         */
    float *p;              /* Read/Write pointer          */
    int k;                  /* Coefficients per polyphase  */
    int l;                  /* Interpolation factor        */
} fir_state;

```

Description

The `fir_interp` function performs a polyphase 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 filter characteristics are dependent upon the number of filter coefficients and their values, and on the interpolation factor supplied by the calling program. Each set of polyphase filter coefficients should be stored continually in reverse order. Therefore, if the filter coefficients have been

Run-Time Library Reference

generated in normal order by a filter design tool, they will have to be re-ordered before they are passed to the filter, as illustrated by the following formula:

```
filter_coeffs [(nc * nphases) - np]
```

where:

```
nc = {1, 2, ..., ncoeffs}
```

```
np = {1, 2, ..., nphases}
```

`ncoeffs` represents the number of coefficients per polyphase filter
`nphases` represents the number of polyphase filters and is set to the interpolation factor

For example, if the number of polyphase filters is 4 and the total number of coefficients is 12, then the coefficients should be ordered as shown below:

```
filter_coeffs[nphases-1],  
filter_coeffs[(2*nphases)-1],  
filter_coeffs[(3*nphases)-1],  
...  
filter_coeffs[nphases-4],  
filter_coeffs[(2*nphases)-4],  
filter_coeffs[(3*nphases)-4].
```

The `fir_interp` function assumes that the filter coefficients are aligned on a quad-word boundary.

A pointer to the coefficients is passed into the `fir_interp` function via the argument `s`, which is a structured variable that represents the filter state. This structured variable must be declared and initialized before calling the function. The `filter.h` header file contains the macro `fir_init` that can be used to initialize the variable and is defined as:

```

#define fir_init(state, coeffs, delay, ncoeffs, index) \
    (state).h = (coeffs);    \
    (state).d = (delay);     \
    (state).p = (delay);     \
    (state).k = (ncoeffs);   \
    (state).l = (index)

```

The interpolation factor is supplied to the function in `s->l`. A pointer to the coefficients should be stored in `s->h`, and `s->k` should be set to the number of coefficients per polyphase filter.

Each filter should have its own delay line which is a vector of type `float` 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`, the read/write pointer, to keep track of its current position within the vector.

The output returned by the function will be multiplied by the number of polyphase filters. Therefore, each element of the output vector will have to be scaled accordingly. This is demonstrated in the **Example** below.

Algorithm

$$y_m(k) = \sum_{i=0}^{p/l-1} x(k-i) * h(i * l + m)$$

where $m=\{0,1,2,\dots,l\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

Run-Time Library Reference

Example

```
#include <filter.h>

#define INTERP_FACTOR 2
#define NCOEFFS      24
#define NSAMPLES     128
#define NPOLY        INTERP_FACTOR

/* Coefficients in normal order */

float filter_coeffs[NCOEFFS];

/* Coefficients in implementation order */

#pragma align 4;
float coeffs[NCOEFFS];

/* Input, Output, Delay Line, and Filter State */

float input[NSAMPLES], output[INTERP_FACTOR*NSAMPLES];
float delay[NCOEFFS];

fir_state state;

/* Utility Variables */

int scale;
int i,nc,np;

/* Transform the normal order coefficients from a filter design
   tool into coefficients for the fir_interp function */

for (np = 1, i = 0; np <= NPOLY; np++)
    for (nc = 1; nc <= (NCOEFFS/NPOLY); nc++)
        coeffs[i++] = filter_coeffs[(nc * NPOLY) - np];

/* Initialize the delay line */

for (i = 0; i < NCOEFFS; i++)
```

```
    delay[i] = 0;

/* Initialize the filter state */

fir_init (state, coeffs, delay, (NCOEFFS/NPOLY), INTERP_FACTOR);

/* Call the fir_interp function */

fir_interp (input, output, NSAMPLES, &state);

/* Adjust output */

scale = NPOLY;
for (i = 0; i < (INTERP_FACTOR*NSAMPLES); i++)
    output[i] = output[i] / scale;
```

floor

floor

Synopsis

```
#include <math.h>
float floorf (float x)
double floor (double x)
long double floord (long double x)
```

Description

This function calculates and rounds down to the next lowest whole number that is less than or equal to number x .

Algorithm

return = largest int < x

Domain

$x = [-3.4 \times 10^{38} \dots 3.4 \times 10^{38}]$	for floorf()
$x = [-1.7 \times 10^{308} \dots 1.7 \times 10^{308}]$	for floord()

fmax

maximum

Synopsis

```
#include <math.h>

float fmaxf (float parm1, float parm2);
double fmax (double parm1, double parm2);
long double fmaxd (long double parm1, long double parm2);
```

Description

The `fmax` functions return the larger of their two arguments.

The `fmaxf` function is a built-in function which is implemented with a `MAX` instruction. The `fmax` function will be compiled as a built-in function if `double` is the same size as `float`.

Algorithm

```
if ( parm1 > parm2 )
    return (parm1)
else
    return (parm2)
```

Domain

Full range.

fmin

minimum

Synopsis

```
#include <math.h>

float fminf (float parm1, float parm2);
double fmin (double parm1, double parm2);
long double fminl (long double parm1, long double parm2);
```

Description

The `fmin` functions return the smaller of their two arguments.

The `fminf` function is a built-in function which is implemented with a `MIN` instruction. The `fmin` function will be compiled as a built-in function if `double` is the same size as `float`.

Algorithm

```
if ( parm1 < parm2 )
    return (parm1)
else
    return (parm2)
```

Domain

Full range.

fmod

floating-point modulus

Synopsis

```
#include <math.h>
```

```
float fmodf (float x, float y)
```

```
double fmod (double x, double y)
```

```
long double fmodd (long double x, long double y)
```

Description

The `fmod` function 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.

Algorithm

$$return = \left(\left\lfloor \frac{x}{y} \right\rfloor - floor \left\lfloor \frac{x}{y} \right\rfloor \right) \cdot sign(x)$$

Domain

$x = [-3.4 \times 10^{38} \dots 3.4 \times 10^{38}]$ for `fmodf()`

$x = [-1.7 \times 10^{308} \dots 1.7 \times 10^{308}]$ for `fmodd()`

NOT $y = 0$

frexp

separate fraction and exponent

Synopsis

```
#include <math.h>
float frexpf (float x, int *n)
double frexp (double x, int *n)
long double frexpd (long double x, int *n)
```

Description

The `frexp` function separates a floating-point input into a normalized fraction and a (base 2) exponent. The function returns the first argument as 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 the fraction and exponent will both be set to zero.

Algorithm

return = f ; n passed through 2nd parameter pointer. $x = f * 2^n$

Domain

$x = [-3.4 \times 10^{38} \dots 3.4 \times 10^{38}]$	for <code>frexpf()</code>
$x = [-1.7 \times 10^{308} \dots 1.7 \times 10^{308}]$	for <code>frexpd()</code>

gen_bartlett

generate bartlett window

Synopsis

```

#include <window.h>
void gen_bartlett(w,a,N)
float 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 $N \cdot a$.

The Bartlett window is similar to the Triangle window (described [on page 3-138](#)) 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.

Run-Time Library Reference

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(w,a,N)
float 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(w,alpha,a,N)
float 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 α is an input parameter

Domain

$a > 0$; $N > 0$; $\alpha > 0.0$

gen_hamming

generate hamming window

Synopsis

```
#include <window.h>
void gen_hamming(w,a,N)
float 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 $N \cdot a$.

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(w,a,N)
float 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 $N \cdot a$. 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(w,a,N)
float 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(w,beta,a,N)
float 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 β 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 $n = \{0, 1, 2, \dots, N-1\}$, $\alpha = (N - 1) / 2$, and $I_0(\beta)$ represents the zeroth-order modified Bessel function of the first kind.

Domain

$\alpha > 0$; $N > 0$; $\beta > 0.0$

gen_rectangular

generate rectangular window

Synopsis

```
#include <window.h>
void gen_rectangular(w,a,N)
float 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 $N \cdot a$.

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(w,a,N)
float 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 . The length of the output vector should therefore be $N \cdot a$.

Refer to the Bartlett window (described [on page 3-129](#)) 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 $n = \{0, 1, 2, \dots, N-1\}$

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(w,a,N)
float 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` (described [on page 3-134](#)).

Domain

$a > 0$; $N > 0$

histogram

histogram

Synopsis

```
#include <stats.h>

void histogramf(a,n,max,min,c,m)
const float a[];          /* Pointer to input vector a */
int n;                    /* Number of input samples */
float max;                 /* Maximum value of the bin */
float min;                 /* Minimum value of the bin */
int c[];                   /* Pointer to output vector c */
int m;                     /* Number of bins */
```

Description

The `histogramf` function computes a histogram of the input vector `a` that contains `n` samples, and stores the result in the output vector `c`.

The minimum and maximum value of any input sample is specified by `min` and `max`, respectively; these values are used by the function to calculate the size of each bin as $(\text{max} - \text{min}) / m$ where `m` is the size of the output vector `c`.

Any input value that is outside the range $[\text{min}, \text{max})$ will exceed the boundaries of the output vector and will be discarded.

Algorithm

The output vector is first zeroed by the function. Each input value is then adjusted by `min`, multiplied by $1/\text{binsize}$, and rounded to select the appropriate bin to increment.

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

ifft

N point inverse FFT

Synopsis

```
#include <filter.h>
void ifft(in[], t[], out[], w[], wst, n)
const complex_float in[];      /* Input sequence          */
complex_float t[];             /* Temporary working buffer */
complex_float out[];           /* Output sequence          */
const complex_float w[];       /* Twiddle sequence         */
int wst;                       /* Twiddle factor stride    */
int n;                         /* Number of FFT points     */
```

Description

This function transforms the frequency domain complex input signal sequence to the time domain by using the accelerated version of the ‘Discrete Fourier Transformation’ known as an ‘Inverse Fast Fourier Transform’ or IFFT. It will “decimate in frequency” by the best choice FFT algorithm, radix4 or mixed-radix, depending on the input sequence length.

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. If the input data can be overwritten, then the memory requirements may be reduced by specifying the input array as the output array. Run-time performance of the function will be improved if the input and output arrays are allocated in a different memory block than the twiddle table, *w*.

The twiddle table is passed in the argument *w*, which must contain at least $\frac{3}{4}n$ complex twiddle factors. The function *twidfft* may be used to initialize the array. If the twiddle table contains more factors than needed for a

particular call on `ifft`, then the stride factor has to be set appropriately; otherwise it should be 1. Refer to [“twidfft” on page 3-212](#) for more information.

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 radix4, or direct mixed radix algorithm. To get the inverse effect, it first swaps the real and imaginary parts of the input, performs the direct radix4 or mixed radix transformation and finally swaps the real and imaginary parts of the output.

Domain

Input sequence length n must equal to either a power of two, or a power of four, and at least 16.

ifft2d

NxN point 2-D inverse input FFT

Synopsis

```
#include <filter.h>
void ifft2d(*in, *t, *out, w[], wst, n)
const complex_float *in; /* Pointer to input matrix a[n][n] */
complex_float *t;        /* Pointer to working buffer t[n][n] */
complex_float *out;      /* Pointer to matrix c[n][n] */
const complex_float w[]; /* Twiddle sequence */
int wst;                 /* Twiddle factor stride */
int n;                   /* Number of FFT points */
```

Description

This function computes a two-dimensional inverse Fast Fourier Transform of the complex input matrix `a[n][n]` and stores the result in the complex matrix `c[n][n]`.

If input data can be overwritten, the optimum memory usage is achieved by setting the output pointer to the input array.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `ifft2d`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 3-212.](#)

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 equal to either a power of two, or a power of four, and at least 16.

iir

infinite impulse response filter

Synopsis

```
#include <filter.h>
void iir(x,y,n,s)
const float x[];      /* Input sample vector x          */
float y[];             /* Output sample vector y          */
int n;                /* Number of input samples         */
iir_state *s;         /* Pointer to filter state structure */
```

The IIR filter 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 biquad stages         */
} iir_state;
```

Description

The `iir` function implements a biquad, 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 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 `iir_init`, in the `filter.h` header file, is available to initialize the structure and is defined as:

```
#define iir_init(state, coeffs, delay, stages) \
    (state).c = (coeffs); \
    (state).d = (delay); \
    (state).k = (stages)
```

The characteristics of the filter are dependent upon the 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 `float` 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.

The `iir` function assumes that the delay line is quad-word aligned. There is no performance penalty if the input data, output vector, and filter state are allocated in the same memory block.



The `iir` function is implemented using a direct form II algorithm. When importing coefficients from a filter design tool that employs a transposed direct form II, the A1 and A2 coefficients have to be negated. For example, if a filter design tool returns $A = [1.0, 0.2, -0.9]$, then the A coefficients have to be modified to $A = [1.0, -0.2, 0.9]$.

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

-3.4×10^{38} to $+3.4 \times 10^{38}$

interrupt, interruptf, interrupts

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);
void (*interruptnr (int sig, void(*func)(int))) (int);
void (*interruptfnr (int sig, void(*func)(int))) (int);
void (*interruptsnr (int sig, void(*func)(int))) (int);
```

Description

The `interrupt` function determines how a signal received during program execution is handled. The `interrupt` function executes the function pointed to by `func` at every interrupt `sig`, whereas the `signal` function executes the function only once.

The `sig` argument must be one of the values that are listed in [Table 3-18 on page 3-172](#) and the `func` argument must be one of the values that are listed in [Table 3-17](#). The `interrupt` function causes the receipt of the signal number `sig` to be handled in one of the following ways:

Table 3-17. Interrupt Handling

Func Value	Action
SIG_DFL	The default action is taken.
SIG_IGN	The signal is ignored.
Function address	The function pointed to by <code>func</code> is executed.

The function pointed to by `func` is executed each time the interrupt is received for handlers installed with `interrupt` (or on the first received signal in the case of handlers installed with `signal`). The `interrupt` function must be called with the `func` argument set to `SIG_IGN` to disable interrupt handling.

The `interrupt`, `interruptf`, and `interrupts` functions perform similar services and always enable nested interrupts, but they differ in the manner in which they dispatch an interrupt. The `interrupt` function is appropriate for interrupt service routines that are written in C or C++. This function uses the normal interrupt dispatcher, which provides the following services:

- Preserves all registers
- Resets the circular buffer base and length registers for linear access
- Preserves static condition flags
- Preserves the data alignment buffers
- Calls the handler function
- Restores state ready for return to the interrupted routine

The `interruptf` function uses a fast interrupt dispatcher that is only suitable for interrupt service routines that have been implemented in assembler. The fast interrupt dispatcher provides the same services as the normal interrupt dispatcher except that it does not preserve or restore any of the following registers:

- The circular buffer base and length registers
- The static condition flags
- The summation registers `PR 1:0`
- The data alignment buffers
- The Enhanced Communication Instruction registers

Run-Time Library Reference

The `interrupts` function uses a super interrupt dispatcher that preserves and restores only the resources used in order to perform the dispatch. For this reason, it should only be used with interrupt service routines that have been written in assembler. The interrupt service routine must preserve all the registers that it uses.

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

The `interrupt`, `interruptf`, and `interrupts` functions install interrupt handlers that can be nested. This means that when an interrupt is being serviced by a routine installed by one of these functions, a higher priority interrupt or signal may be serviced before the routine has finished dealing with the initial interrupt.

The `interruptnr`, `interruptfnr`, and `interruptsnr` functions install non-reentrant interrupt handlers, meaning that on servicing an interrupt, interrupts and signals are disabled until the execution of the service routine has completely finished.

It is possible to mix both reentrant and non-reentrant interrupt and signal handlers in the same project at the expense of the additional code space used by the different dispatchers.

Error Conditions

The `interrupt` function returns `SIG_ERR` and sets `errno` to a positive non-zero value if it does not recognize the requested signal.

Example

```
#include <signal.h>

interrupt (SIGIRQ2, irq2_handler);
/* enable hardware interrupt pin 2 handler */
```

```
interrupt (SIGIRQ2, SIG_IGN);  
/* disable hardware interrupt pin 2 handler */
```

See Also

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

log

natural logarithm

Synopsis

```
#include <math.h>
float logf (float x)
double log (double x)
long double logd (long double x)
```

Description

This function calculates the natural (base e) logarithm of number x .

If x equals 0, this function will return -3.4×10^{38} for a `float` return value and -1.7×10^{308} for a `long double` return value.

Algorithm

return = $\ln(x)$

Domain

$x = [0.0 \dots 3.4 \times 10^{38}]$ for `logf()`

$x = [0.0 \dots 1.7 \times 10^{308}]$ for `logd()`

log10

base 10 logarithm

Synopsis

```
#include <math.h>
float log10f (float x)
double log10 (double x)
long double log10d (long double x)
```

Description

This function calculates the base 10 logarithm of number x .

If x equals 0, this function will return -3.4×10^{38} for a `float` return value and -1.7×10^{308} for a `long double` return value.

Algorithm

return = log(x)

Domain

$x = [0.0 \dots 3.4 \times 10^{38}]$ for `log10f()`

$x = [0.0 \dots 1.7 \times 10^{308}]$ for `log10d()`

matinv

matrix inversion

Synopsis

```
#include <matrix.h>
float *matinvf(a,n,c)
const float *a;      /* Pointer to input matrix a[][] */
int n;               /* Number of rows in matrix a[][] */
float *c;            /* Pointer to output matrix c[][] */
```

Description

This function computes the inverse of input matrix `a[][]` and stores the result in the output matrix `c[][]`. The dimensions of matrix `a` and matrix `c` are `n` by `n`. If an inverse of the input matrix exists, a pointer to the output matrix is returned; if no inverse exists the function will return a null pointer.

Algorithm

The function employs Gauss-Jordan elimination with full pivoting.

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matmadd

matrix + matrix addition

Synopsis

```

#include <matrix.h>
void matmaddf(a,b,n,m,c)
const float *a;    /* Pointer to input matrix a[][]          */
const float *b;    /* Pointer to input matrix b[][]          */
int n;             /* Number of rows in matrix a[][] and b[][] */
int m;             /* Number of columns in matrix a[][] and b[][] */
float *c;          /* Pointer to matrix c[][]                */

```

Description

This function adds the input matrix $a[][]$ to the input matrix $b[][]$ placing the result into the output matrix $c[][]$. The dimensions of matrix $a[][]$ are n and m and the dimensions of matrix b are n and m . The resulting matrix $c[][]$ is of dimensions n and m .

Algorithm

$$c_{i,j} = a_{i,j} + b_{i,j}$$

where $i=\{0,1,2,\dots,n-1\}$, $j=\{0,1,2,\dots,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matmmlt

matrix * matrix multiplication

Synopsis

```
#include <matrix.h>
void matmmltf(a,n,k,b,m,c)
const float *a;          /* Pointer to input matrix a[][] */
int n;                   /* Number of rows in matrix a[][] */
int k;                   /* Number of columns in matrix a[][] */
const float *b;          /* Pointer to input matrix b[][] */
int m;                   /* Number of columns in matrix b[][] */
float *c;                 /* Pointer to matrix c[][] */
```

Description

This function computes the multiplication of input matrix $a[][]$ with input matrix $b[][]$, and stores the result to matrix $c[][]$. The dimensions of matrix $a[][]$ are n and k and the dimensions of matrix b are k and m . The resulting matrix $c[][]$ is of dimensions n and m .

Algorithm

$$c_{i,j} = \sum_{l=0}^{k-1} a_{i,l} * b_{l,j}$$

where $i=\{0,1,2,...,n-1\}$, $j=\{0,1,2,...,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matmsub

matrix - matrix subtraction

Synopsis

```

#include <matrix.h>
void matmsubf(a,b,n,m,c)
const float *a;    /* Pointer to input matrix a[][]          */
const float *b;    /* Pointer to input matrix b[][]          */
int n;             /* Number of rows in matrix a[][] and b[][] */
int m;             /* Number of columns in matrix a[][] and b[][] */
float *c;          /* Pointer to matrix c[][]                */

```

Description

This function computes the subtraction of input matrix $a[][]$ with input matrix $b[][]$, and stores the result to matrix $c[][]$. The dimensions of matrix $a[][]$ are n and m and the dimensions of matrix b are n and m . The resulting matrix $c[][]$ is of dimensions n and m .

Algorithm

$$c_{i,j} = a_{i,j} - b_{i,j}$$

where $i=\{0,1,2,...,n-1\}$, $j=\{0,1,2,...,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matsadd

matrix + scalar addition

Synopsis

```
#include <matrix.h>
void matsaddf(a,b,n,m,c)
const float *a;    /* Pointer to input matrix a[][] */
float b;           /* Value of input scalar b */
int n;             /* Number of rows in matrix a[][] */
int m;             /* Number of columns in matrix a[][] */
float *c;          /* Pointer to matrix c[][] */
```

Description

This function computes the addition of input matrix `a[][]` with input scalar `b`, and stores the result to matrix `c[][]`. The dimensions of matrix `a[][]` are `n` and `m`. The resulting matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$c_{i,j} = a_{i,j} + b$$

where $i=\{0,1,2,\dots,n-1\}$, $j=\{0,1,2,\dots,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matmult

matrix * scalar multiplication

Synopsis

```
#include <matrix.h>
void matmultf(a,n,k,b,m,c)
const float *a;      /* Pointer to input matrix a[][] */
float b;              /* Value of input scalar b */
int n;                /* Number of rows in matrix a[][] */
int m;                /* Number of columns in matrix a[][] */
float *c;              /* Pointer to matrix c[][] */
```

Description

This function computes the multiplication of input matrix `a[][]` with input scalar `b`, and stores the result to matrix `c[][]`. The dimensions of matrix `a[][]` are `n` and `m`. The resulting matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$c_{i,j} = a_{i,j} * b$$

where $i=\{0,1,2,\dots,n-1\}$, $j=\{0,1,2,\dots,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

matssub

matrix - scalar subtraction

Synopsis

```
#include <matrix.h>
void matssubf(a,b,n,m,c)
const float *a;      /* Pointer to input matrix a[][] */
float b;              /* Value of input scalar b */
int n;                /* Number of rows in matrix a[][] */
int m;                /* Number of columns in matrix a[][] */
float *c;              /* Pointer to matrix c[][] */
```

Description

This function computes the subtraction of input matrix `a[][]` with input scalar `b`, and stores the result to matrix `c[][]`. The dimensions of matrix `a[][]` are `n` and `m`. The resulting matrix `c[][]` is of dimensions `n` and `m`.

Algorithm

$$c_{i,j} = a_{i,j} - b$$

where $i=\{0,1,2,\dots,n-1\}$, $j=\{0,1,2,\dots,m-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

max

maximum

Synopsis

```
#include <stdlib.h>

int max (int parm1, int parm2)
long int lmax (long int parm1, long int parm2);
long long int llmax (long long int parm1, long long int parm2);
```

Description

The max functions return the larger of their two arguments. The functions are built-in functions that are implemented with a MAX instruction.

Algorithm

```
if (parm1 > parm2)
    return(parm1)
else
    return(parm2)
```

Domain

Full range for type of parameters used.

mean

mean

Synopsis

```
#include <stats.h>
float meanf(a,n)
const float a[];      /* Input vector a          */
int n;                /* Number of input samples */
```

Description

This function computes the mean of the n elements contained within input vector a and returns the result.

Algorithm

$$c = \frac{1}{n} * \left(\sum_{i=0}^{n-1} a_i \right)$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

min

minimum

Synopsis

```
#include <stdlib.h>

int min (int parm1, int parm2)
long int lmin (long int parm1, long int parm2);
long long int llmin (long long int parm1, long long int parm2);
```

Description

The min functions return the smaller of their two arguments. The functions are built-in functions that are implemented with a `MIN` instruction.

Algorithm

```
if (parm1 < parm2)
    return(parm1)
else
    return(parm2)
```

Domain

Full range for type of parameters used.

modf

separate integral and fractional parts

Synopsis

```
#include <math.h>
double modf(double f, double *fraction);
float modff (float f, float *fraction);
long double modfd (long double f, long double *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.

Algorithm

$$\text{return} = (|f| - \text{floor}|f|) \cdot \text{sign}(f)$$
$$\text{fraction} = (\text{floor}|f| \cdot \text{sign}(f))$$

Domain

$f = [-3.4 \times 10^{38} \dots 3.4 \times 10^{38}]$ for `modf()`

$f = [-1.7 \times 10^{308} \dots 1.7 \times 10^{308}]$ for `modfd()`

mu_compress

μ-law compression

Synopsis

```

#include <filter.h>
void mu_compress(in, out, n)
const int in[];      /* Input array */
int 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`. The function has been optimized and requires that both the input and output vectors are quad-word aligned.

Algorithm

$$C(k) = \mu_law \text{ compression of } A(k) \text{ for } k=0 \text{ to } n-1.$$

Domain

Content of input array: -8192 to 8191

mu_expand

μ-law expansion

Synopsis

```
#include <filter.h>
void mu_expand(in, out, n)
const int in[];      /* Input array */
int out[];           /* Output array */
int n;               /* Number of elements to be compressed */
```

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 by `out`. The function has been optimized and requires that both the input and output vectors are quad-word aligned.

Algorithm

$C(k) = \mu_law \text{ expansion of } A(k) \text{ for } k=0 \text{ to } n-1$

Domain

Content of input array: 0 to 255

norm

normalization

Synopsis

```
#include <complex.h>
complex_float normf(complex_float a);
complex_double norm(complex_double a);
complex_long_double normd (complex_long_double a);
```

Description

These functions normalize the complex input *a* and return 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 <code>normf()</code>
-1.7×10^{308} to 1.7×10^{308}	for <code>normd()</code>

polar

construct from polar coordinates

Synopsis

```
#include <complex.h>
complex_float polarf(float mag, float phase);
complex_double polar (double mag, double phase);
complex_long_double polard (long double mag, long double phase);
```

Description

These functions transform the polar coordinate, specified by the arguments `mag` and `phase`, into a Cartesian coordinate and return the result as a complex number in which the x-axis is represented by the real part, and the y-axis by the imaginary part. The `phase` argument is interpreted as radians.

Algorithm

$$\text{Re}(c) = r \cdot \cos(\theta)$$

$$\text{Im}(c) = r \cdot \sin(\theta)$$

where θ is the phase, and r is the magnitude

Domain

phase = [-1,647,095 ... 1,647,095] for `polarf()`
mag = -3.4×10^{38} to $+3.4 \times 10^{38}$

phase = [-843,314,850 ... 843,314,850] for `polard()`
mag = -1.7×10^{308} to $+1.7 \times 10^{308}$

pow

raise to a power

Synopsis

```
#include <math.h>
float powf (float x, float y)
double pow (double x, double y)
long double powd (long double x, long double y)
```

Description

This function calculates x to the power y . If $x < 0$ and y is not an integral value, this function returns 0. If $x = 0$ and $y = 0$, this function returns 0. If overflow occurs, this function returns 3.4×10^{38} for a float-type return value and 1.7×10^{308} for a double-type return value; if underflow occurs, this function returns -3.4×10^{38} for a float return value and -1.7×10^{308} for a long double return value.

Algorithm

return = x^y

Domain

$x, y = [-3.4 \times 10^{38} \dots 3.4 \times 10^{38}]$
 except $x < 0, y \neq i$, where i is an integer for `powf()`

$x, y = [-1.7 \times 10^{308} \dots 1.7 \times 10^{308}]$
 except $x < 0, y \neq i$, where i is an integer for `powd()`

qsort

quick sort

Synopsis

```
#include <stdlib.h>
void qsort(void base, size_t nelem, size_t size,
           int (*compare) (const void *, const void *));
```

Description

The `qsort` function sorts an array of `nelem` objects, pointed to by `base`. 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 pre-sorted array. Note that:

- `base` points to the start of the array
- `nelem` is the number of elements in the array
- `size` is the size of each element of the array
- `compare` points to the function used to compare two elements. It is passed two arguments that point to the array elements being compared. The function 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.

Algorithm

See “Description”.

Domain

N/A

raise

force a signal

Synopsis

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

Description

The `raise` function sends the signal `sig` to the executing program. The `raise` function forces interrupts wherever possible and simulates an interrupt otherwise. The `sig` argument must be one of the signals listed in priority order in [Table 3-18](#). See the hardware reference manual for the target TigerSHARC processor for further details on the hardware interrupts.

Table 3-18. Raise Function Signals—Values and Meanings

Sig Value	Definition
SIGTIMER0LP	Timer 0 low priority interrupt
SIGTIMER1LP	Timer 1 low priority interrupt
SIGLINK0	Link port 0 interrupt
SIGLINK1	Link port 1 interrupt
SIGLINK2	Link port 2 interrupt
SIGLINK3	Link port 3 interrupt
SIGDMA0	DMA channel 0 interrupt
SIGDMA1	DMA channel 1 interrupt
SIGDMA2	DMA channel 2 interrupt
SIGDMA3	DMA channel 3 interrupt
SIGDMA4	DMA channel 4 interrupt

Table 3-18. Raise Function Signals—Values and Meanings (Cont'd)

Sig Value	Definition
SIGDMA5	DMA channel 5 interrupt
SIGDMA6	DMA channel 6 interrupt
SIGDMA7	DMA channel 7 interrupt
SIGDMA8	DMA channel 8 interrupt
SIGDMA9	DMA channel 9 interrupt
SIGDMA10	DMA channel 10 interrupt
SIGDMA11	DMA channel 11 interrupt
SIGDMA12	DMA channel 12 interrupt
SIGDMA13	DMA channel 13 interrupt
SIGIRQ0	Hardware interrupt pin 0 interrupt
SIGIRQ1	Hardware interrupt pin 1 interrupt
SIGIRQ2	Hardware interrupt pin 2 interrupt
SIGIRQ3	Hardware interrupt pin 3 interrupt
SIGBUSLK	Bus lock interrupt
SIGTIMER0HP	Timer 0 high priority interrupt
SIGTIMER1HP	Timer 1 high priority interrupt
SIGHW	Hardware error interrupt
SIGSW	Software exception
SIGDBG	Emulation debug interrupt
SIGABRT	Standard abort signal
SIGILL	Standard illegal function signal
SIGINT	Standard interactive attention signal
SIGSEGV	Standard illegal storage access signal

Table 3-18. Raise Function Signals—Values and Meanings (Cont'd)

Sig Value	Definition
SIGTERM	Standard termination request signal
SIGFPE	Standard arithmetic error signal

Error Conditions

The `raise` function returns a zero if successful, a non-zero value if it fails.

Example

```
#include <signal.h>
raise (SIGIRQ2);
    /* invoke the hardware interrupt pin 2 handler */
```

See Also

[interrupt](#), [interruptf](#), [interrupts](#), [signal](#), [signalf](#), [signals](#)

rand

random number generator

Synopsis

```
#include <stdlib.h>
int rand (void)
```

Description

This function returns a pseudo-random integer value in the range $[0, 2^{31} - 1]$.

For this function, the measure of randomness is its periodicity, the number of values it is likely to generate before repeating a pattern. The output of the pseudo-random number generator has a period in the order of $2^{31} - 1$.

Algorithm

The algorithm is based on a linear congruential generator.

Domain

N/A

rfft

N point real input FFT

Synopsis

```
#include <filter.h>
```

```
void rfft(in[], t[], out[], w[], wst, n)
const float in[];           /* Input sequence          */
complex_float t[];          /* Temporary working buffer */
complex_float out[];        /* Output sequence         */
const complex_float w[];    /* Twiddle sequence        */
int wst;                    /* Twiddle factor stride   */
int n;                      /* Number of FFT points    */
```

Description

This function transforms the time domain real input signal sequence to the frequency domain by using the accelerated version of the ‘Discrete Fourier Transformation’ known as a ‘Fast Fourier Transform’ or FFT. It will “decimate in frequency” by the best choice FFT algorithm, radix-4 or mixed-radix, depending on the input sequence length. At the initial stage of the transformation, the `rfft` 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. If the input data can be overwritten, then the memory requirements may be reduced by specifying the input array as the output array provided that the size of the input array is at least $2*n$. Run-time performance of the function will be improved if the input and output arrays are allocated in a different memory block than the twiddle table, `w`.

The twiddle table is passed in the argument `w`, which must contain at least $\frac{3}{4}n$ complex twiddle factors. The function `twidfft` may be used to initialize the array. If the twiddle table contains more factors than needed for a particular call on `rfft`, then the stride factor has to be set appropriately; otherwise it should be 1. [For more information, see “twidfft” on page 3-212.](#)

i The library also contains the `rfftf` function ([on page 3-183](#)), which is an optimized implementation of a fast radix-2 algorithm. The characteristics of the FFT generated by the function differ to some extent from that generated by the `rfft` function. Also, for optimal performance, the `rfftf` function makes certain assumptions concerning the alignment of the input, output, and twiddle arrays.

Algorithm

See [“cfft” on page 3-74](#).

Domain

Input sequence length `n` must equal to either a power of two, or a power of four, and at least 16.

Example

```
/* Example to demonstrate how to generate two real FFTs using
   a single twiddle table */

#include <filter.h>

#define NDATA1 256
#define NDATA2 32

float data1[NDATA1];      /* data for a 256-point FFT */
float data2[NDATA2];      /* data for a 32-point FFT */
```

Run-Time Library Reference

```
complex_float output1[NDATA1];
complex_float output2[NDATA2];

static complex_float twidtab[(3*NDATA1)/4];
complex_float temp[NDATA1];
    /* note that the temporary buffer should be as large as the
       largest FFT generated */

/* Generate a twiddle table for a 256-point FFT */

twidfft (twidtab, NDATA1);
    /* note that a twiddle table is constant for a given
       number of FFT points */

/* Generate a 256-point real FFT */

rfft (data1, temp, output1, twidtab, 1, NDATA1);
    /* note that the twiddle table stride factor is 1 */

/* Generate a 32-point real FFT */

rfft (data2, temp, output2, twidtab, (NDATA1/NDATA2), NDATA2);
    /* note that the twiddle table stride factor is 8 */
```

rfft_mag

rfft magnitude

Synopsis

```
#include <filter.h>

float rfft_mag (const complex_float dm input[],
               float dm output[],
               int fftsize);
```

Description

The `rfft_mag` function computes a normalized power spectrum from the output signal generated by a `rfftN` function. Due to the symmetry of a real FFT about the midpoint, this function generates the power spectrum using only the first `fftsize/2` values in the signal. The size of the signal and the size of the power spectrum is `fftsize/2`.



The Nyquist frequency is located at `fftsize/2`.

Algorithm

$$\text{magnitude}(z) = \frac{2\sqrt{\text{Re}(z)^2 + \text{Im}(z)^2}}{\text{fftsize}}$$

Example

```
#include <filter.h>
#define N 64

float fft_input[N];
complex_float fft_output[N/2];

float fft_temp[N];
complex_float twid[(3*N)/4];
```

Run-Time Library Reference

```
float spectrum[N/2];

/* Generate a twiddle table for the FFT */

twidfft twid, N);
    /* Note that a twiddle table is constant for a given number
       of FFT points */

/* Generate a real FFT using rfft */

rfft (fft_input, temp, fft_output, twid, 1, N);

/* Generate a power spectrum

rfft_mag (fft_output, spectrum, N);
```

rfft2d

NxN point 2-D real input FFT

Synopsis

```

#include <filter.h>
void rfft2d(*in, *t, *out, w[], wst, n)
const float *in;           /* Pointer to input matrix a[n][n] */
complex_float *t;          /* Pointer to working buffer t[n][n] */
complex_float *out;        /* Pointer to matrix c[n][n] */
const complex_float w[];   /* Twiddle sequence */
int wst;                   /* Twiddle factor stride */
int n;                    /* Number of FFT points */

```

Description

This function computes a two-dimensional Fast Fourier Transform of the real input matrix `a[n][n]`, and stores the result in the complex matrix `c[n][n]`.

If input data can be overwritten, the optimum memory usage is achieved by setting the output pointer to the input array provided that the memory size of the input array is at least twice $n*n$.

For efficiency, the “twiddle table” is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. An initialization function, `twidfft`, is provided.

If the twiddle table has been allocated at a larger size than needed for a particular call of `rfft2d`, then the stride parameter will need to be set appropriately; otherwise, it should be one. [For more information, see “twidfft” on page 3-212.](#)

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 equal to either a power of two, or a power of four, and at least 16.

rfftf

fast N point real input FFT

Synopsis

```
#include <filter.h>
void rfftf(in[], out[], twid[], wst, n)
const float in[];           /* Input sequence          */
complex_float out[];        /* Output sequence         */
const complex_float twid[]; /* Twiddle sequence        */
int wst;                    /* Twiddle factor stride   */
int n;                     /* Number of FFT points    */
```

Description

The `rfftf` function transforms the time domain real input signal sequence to the frequency domain by using the accelerated version of the ‘Discrete Fourier Transform’ known as a ‘Fast Fourier Transform’ or FFT. It will “decimate in frequency” using an optimized radix-2 algorithm.

The size of the input array `in` is `n`, where `n` represents the number of points in the FFT. The `rfftf` function has been designed for optimum performance and requires that the input array `in` be aligned on an address boundary that is a multiple of the FFT size. For certain applications, this alignment constraint may not be appropriate and, in such cases, the application should call the `rfft` function (see [on page 3-176](#)) instead (with a consequent loss of some performance).

Due to the symmetry of a real FFT, only $n/2$ points of the output are computed by the function and they are stored in the output array `out`. Also, each value of the output will be double the actual FFT value. This behavior is in contrast to that of the `rfft` function, which generates all `n` points of the FFT and does not double the magnitude of the FFT output.

The twiddle table is passed in the argument `twid`, which must contain at least $n/2$ complex twiddle factors. The function `twidfft` may be used to initialize the array. If the twiddle table contains more factors than required

Run-Time Library Reference

for a particular FFT size, then the stride factor `wst` has to be set appropriately; otherwise, it should be set to 1. [For more information, see “twidfft” on page 3-214.](#)



The twiddle tables used by the functions `rfft` and `rfftf` are not compatible. The `rfft` function uses a twiddle table that contains $\frac{3}{4}n$ factors in which the imaginary coefficients are positive sine values, while the `rfftf` function uses a twiddle table with $\frac{1}{2}n$ factors in which the imaginary coefficients are negative sine values.

It is recommended that the twiddle table and the output array are allocated in separate memory blocks—otherwise, the performance of the function will degrade.

Algorithm

See [“cfft” on page 3-81.](#)

Domain

The number of points in the FFT must be a power of 2 and must be at least 64.

rfftf_mag

rfftf magnitude

Synopsis

```
#include <filter.h>

float rfftf_mag (const complex_float dm input[],
                 float dm output[],
                 int fftsize);
```

Description

The `rfftf_mag` function computes a normalized power spectrum from the output signal generated by a `rfftf` function. The size of the signal and the size of the power spectrum is `fftsize/2`.



The Nyquist frequency is located at `fftsize/2`.

Algorithm

$$\text{magnitude}(z) = \frac{\sqrt{\text{Re}(z)^2 + \text{Im}(z)^2}}{\text{fftsize}}$$

Example

```
#include <filter.h>
#define N 64

#pragma align 64
section ("data2") float fft_input[N];
section ("data1") complex_float fft_output[N/2];
section ("data1") complex_float twiddle_table[N/2];

float spectrum[N/2];

/* Generate a twiddle table for the FFT */
```

Run-Time Library Reference

```
twidfft twid_table, N);  
    /* Note that a twiddle table is constant for a given number  
       of FFT points */  
  
/* Generate a real FFT using rfft */  
  
rfft (fft_input, fft_output, twid_table, 1, N);  
  
/* Generate a power spectrum  
  
rfft_mag (fft_output, spectrum, N);
```

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 */
```

Description

This function computes the root mean square of the input elements contained within input vector *a* and returns the result.

Algorithm

$$c = \sqrt{\frac{\sum_{i=0}^{n-1} a_i^2}{n}}$$

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸

rsqrt

reciprocal square root

Synopsis

```
#include <math.h>
float rsqrtf (float x)
double rsqrt (double x)
long double rsqrtld (long double x)
```

Description

This function calculates the reciprocal of the square root of the number x .
If x is negative, the function returns 0.

Algorithm

$$return = 1/(\sqrt{x})$$

Domain

$x = [0.0 \text{ to } +3.4 \times 10^{38}]$ for `rsqrtf()`

$x = [0.0 \dots 1.7 \times 10^{308}]$ for `rsqrtld()`

sign

sign

Synopsis

```
#include <math.h>
float signf (float parm1, float parm2)
double sign (double parm1, double parm2)
long double signl (long double parm1, long double parm2)
```

Description

This function copies the sign of the second argument to the first argument.

Algorithm

```
return( |parm1| * signof( parm2))
```

Domain

Full range

signal, signalf, signals

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);
void (*signalnr (int sig, void(*func)(int))) (int);
void (*signalfnr (int sig, void(*func)(int))) (int);
void (*signalsnr (int sig, void(*func)(int))) (int);
```

Description

The `signal` function determines how a signal received during program execution is handled. The function sets up a handler that can respond to a single occurrence of an interrupt. It returns the value of the previously installed interrupt or signal handler action.

The `sig` argument must be one of the values that are listed in [Table 3-18 on page 3-172](#) and the `func` argument must be one of the values that are listed in [Table 3-19](#). The `signal` function causes the receipt of the signal number `sig` to be handled in one of the following ways:

Table 3-19. Interrupt Handling

Func Value	Action
SIG_DFL	The default action is taken.
SIG_IGN	The signal is ignored.
Function address	The function pointed to by <code>func</code> is executed.

The function pointed to by `func` is executed once when the interrupt is received. Handling of the interrupt is then returned to the default state.

The `signal`, `signal_f`, and `signals` functions perform similar services and always enable nested interrupts, but they differ in the manner in which they dispatch an interrupt. The `signal` function is appropriate for interrupt service routines that are written in C or C++. This function uses the normal interrupt dispatcher, which provides the following services:

- Preserves all registers
- Resets the circular buffer base and length registers for linear access
- Preserves static condition flags
- Preserves the data alignment buffers
- Calls the handler function
- Restores state ready for return to the interrupted routine

The `signal_f` function uses a fast interrupt dispatcher that is only suitable for interrupt service routines that have been implemented in assembler. The fast interrupt dispatcher provides the same services as the normal interrupt dispatcher except that it does not preserve or restore any of the following registers:

- The circular buffer base and length registers
- The static condition flags
- The summation registers `PR 1:0`
- The data alignment buffers
- The Enhanced Communication Instruction registers

The `signals` function uses a super interrupt dispatcher that preserves and restores only the resources used in order to perform the dispatch. For this reason, it should only be used with interrupt service routines that have been written in assembler. The interrupt service routine must preserve all the registers that it uses.

Run-Time Library Reference

The `signal`, `sigalnf`, and `signals` functions install signal handlers that can be nested. This means that when a signal is being serviced by a routine installed by one of these functions, a higher priority signal or interrupt may be serviced before the routine has finished dealing with the initial signal.

The `signalnr`, `signalfnr`, and `signalnr` functions install non-reentrant signal handlers, meaning that on servicing a signal, signals and interrupts are disabled until the execution of the service routine has completely finished.

Error Conditions

The `signal` function returns `SIG_ERR` and sets `errno` to a positive non-zero value if it does not recognize the requested signal.

Example

```
#include <signal.h>

signal (SIGIRQ2, irq2_handler);
/* enable hardware interrupt pin 2 handler */

signal (SIGIRQ2, SIG_IGN);
/* disable hardware interrupt pin 2 handler */
```

See Also

[interrupt](#), [interruptf](#), [interrupts](#), [raise](#)

sin

sine

Synopsis

```
#include <math.h>
float sinf (float x)
double sin (double x)
long double sind (long double x)
```

Description

The `sin` function returns the sine of the argument. The input is interpreted as a radian; the output is in the range $[-1, 1]$. If x is outside of the domain, this function returns 0.0.

Algorithm

return = *sin(x)*

Domain

$x = [-1,647,095 \dots 1,647,095]$	for <code>sinf()</code>
$x = [-843,314,850 \dots 843,314,850]$	for <code>sind()</code>

sinh

hyperbolic sine

Synopsis

```
#include <math.h>
float sinhf (float x)
double sinh (double x)
long double sinhd (long double x)
```

Description

The `sinh` function returns the hyperbolic sine of the argument `x`, where `x` is measured in radians. If `x` is outside the domain, this function returns 3.4×10^{38} for a `float`-type return value and 1.7×10^{308} for a `double`-type return value.

Algorithm

return = sinh(x)

Domain

$x = [-(\ln(3.4 \times 10^{38}) - \ln(2)) \dots (\ln(3.4 \times 10^{38}) - \ln(2))]$	for <code>sinhf()</code>
$x = [-(\ln(1.7 \times 10^{308}) - \ln(2)) \dots (\ln(1.7 \times 10^{308}) - \ln(2))]$	for <code>sinhd()</code>

sqrt

square root

Synopsis

```
#include <math.h>
float sqrtf (float x)
double sqrt (double x)
long double sqrtl (long double x)
```

Description

The `sqrt` function returns the positive square root of the argument `x`.
If `x` is negative, this function returns 0.

Algorithm

return = $\sqrt{x}$

Domain

$x = [0.0 \dots 3.4 \times 10^{38}]$	for <code>sqrtf()</code>
$x = [0.0 \dots 1.7 \times 10^{308}]$	for <code>sqrtl()</code>

srand

random number seed

Synopsis

```
#include <stdlib.h>
void srand(unsigned int newseed)
```

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.

Algorithm

generator_seed = newseed

Domain

0 to RAND_MAX

strtod

convert string to double

Synopsis

```
#include <stdlib.h>
double strtod (const char *nptr, char **endptr)
```

Description

The `strtod` function converts a character string to a `double` value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `nptr` argument is a pointer to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (`+`) or minus (`-`); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (`.`).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:.

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix 0x or 0X. This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter p or P, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens [hexdigs] [.hexdigs].

The first character that does not fit either form of number will stop 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`.

Domain

The number should fit within the dynamic range of a `double`. The function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `HUGE_VAL` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

strtouf

convert string to float

Synopsis

```
#include <stdlib.h>
float strtouf (const char *string, char **end)
```

Description

The `strtouf` function converts a character string to a single-precision floating-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `nptr` argument is a pointer to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (`+`) or minus (`-`); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (`.`).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix 0x or 0X. This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter p or P, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens [hexdigs] [.hexdigs].

The first character that does not fit either form of number will stop 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`.

Domain

The number should fit within the dynamic range of a `float`. The function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate) `FLT_MAX` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

strtoi

convert string to integer

Synopsis

```
#include <stdlib.h>
int strtoi (const char *string, char **end, int base)
```

Description

The `strtoi` function converts a character string to a integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

```
[whitespace] [sign] [base] digits
```

where *whitespace* can consist of spaces and/or tab characters, *sign* is either plus (+) or minus (-), *base* is 0 for octal and 0x for hexadecimal, and *digits* are one or more decimal digits (or letters from a to f for hexadecimal numbers). The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets **end* to that location, provided that *end* is not a null pointer.

If the third argument *base* is zero, then the function tries to determine the base from the information it finds inside the string (see format description above). If the third argument *base* is non-zero, then it has to be between 2 and 36, inclusively. If the third argument *base* does not correspond to any of the above values, it is set to 10. If the third argument *base* is between 11 and 36 (inclusively) the letters of the alphabet from a to z are used as needed to represent the extra digits in the corresponding base.

Domain

-2,147,483,648 to 2,147,483,647

strtol

convert string to long integer

Synopsis

```
#include <stdlib.h>
long strtol (const char *string, char **end, int base)
```

Description

The `strtol` function converts a character string to a integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

[*whitespace*] [*sign*] [*base*] *digits*

where *whitespace* can consist of spaces and/or tab characters, *sign* is either plus (+) or minus (-), *base* is 0 for octal and 0x for hexadecimal, and *digits* are one or more decimal digits (or letters from a to f for hexadecimal numbers). The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets **end* to that location, provided that *end* is not a null pointer.

If the third argument *base* is zero, then the function tries to determine the base from the information it finds inside the string (see format description above). If the third argument *base* is non-zero, then it has to be between 2 and 36, inclusively. If the third argument *base* does not correspond to any of the above values, it is set to 10. If the third argument *base* is between 11 and 36 (inclusively) the letters of the alphabet from a to z will be used as needed to represent the extra digits in the corresponding base.

Domain

-2,147,483,648 to 2,147,483,647

strtold

convert string to long double

Synopsis

```
#include <stdlib.h>
long double strtold (const char *string, char **end)
```

Description

The `strtold` function converts a character string to a double-precision floating-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

```
[whitespace] [sign] [digits] [.digits] [{e|E} [sign] digits]
```

where *whitespace* can consist of spaces and/or tab characters, *sign* is either plus (+) or minus (-), and *digits* are one or more decimal digits. If no digits appear before the decimal point then at least one must appear after the decimal point. The decimal digits may be followed by an exponent denoted by one of the following characters: `e` or `E`, followed by a decimal integer which may be signed.

The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets `*end` to that location, provided that `end` is not a null pointer.

Domain

The number should fit within the dynamic range of a `long double`. The function returns a zero if no conversion could be made. If the correct value results in an overflow, a positive or negative (as appropriate)

Run-Time Library Reference

`LDBL_MAX` is returned. If the correct value results in an underflow, 0.0 is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

strtoll

convert string to long long integer

Synopsis

```
#include <stdlib.h>
long long strtoll (const char *string, char **end, int base)
```

Description

The `strtoll` function converts a character string to a integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

```
[whitespace] [sign] [base] digits
```

where *whitespace* can consist of spaces and/or tab characters, *sign* is either plus (+) or minus (-), *base* is 0 for octal and 0x for hexadecimal, and *digits* are one or more decimal digits (or letters from a to f for hexadecimal numbers). The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets **end* to that location, provided that *end* is not a null pointer.

If the third argument *base* is zero, then the function tries to determine the base from the information it finds inside the string (see format description above). If the third argument *base* is non-zero, then it has to be between 2 and 36, inclusively. If the third argument *base* does not correspond to any of the above values, it is set to 10. If the third argument *base* is between 11 and 36 (inclusively) the letters of the alphabet from a to z will be used as needed to represent the extra digits in the corresponding base.

Domain

-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

strtoul

convert string to unsigned long integer

Synopsis

```
#include <stdlib.h>
unsigned long strtoul (const char *string, char **end, int base)
```

Description

The `strtoul` function converts a character string to a integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

```
[ whitespace ] [ sign ] [ base ] digits
```

where *whitespace* can consist of spaces and/or tab characters, *sign* is either plus (+) or minus (-), *base* is 0 for octal and 0x for hexadecimal, and *digits* are one or more decimal digits (or letters from a to z for hexadecimal numbers). The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets **end* to that location, provided that *end* is not a null pointer.

If the third argument *base* is zero, then the function tries to determine the base from the information it finds inside the string (see format description above). If the third argument *base* is non-zero, then it has to be between 2 and 36, inclusively. If the third argument *base* does not correspond to any of the above values, it is set to 10. If the third argument *base* is between 11 and 36 (inclusively) the letters of the alphabet from a to z will be used as needed to represent the extra digits in the corresponding base.

Domain

0 to 4,294,967,295

strtoull

convert string to unsigned long long integer

Synopsis

```
#include <stdlib.h>
unsigned long long strtoull (const char *string, char **end,
                             int base);
```

Description

The `strtoull` function converts a character string to a integer fixed-point value where the input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

Algorithm

The `string` argument is as follows:

[*whitespace*] [*sign*] [*base*] *digits*

where *whitespace* can consist of spaces and/or tab characters, *sign* is either plus (+) or minus (-), *base* is 0 for octal and 0x for hexadecimal, and *digits* are one or more decimal digits (or letters from a to z for hexadecimal numbers). The function stops reading the input string at the first character that it cannot recognize as part of a valid argument defined above. It sets **end* to that location, provided that *end* is not a null pointer.

If the third argument *base* is zero, then the function tries to determine the base from the information it finds inside the string (see format description above). If the third argument *base* is non-zero, then it has to be between 2 and 36, inclusively. If the third argument *base* does not correspond to any of the above values, it is set to 10. If the third argument *base* is between 11 and 36 (inclusively) the letters of the alphabet from a to z will be used as needed to represent the extra digits in the corresponding base.

Run-Time Library Reference

Domain

0 to 18,446,744,073,709,551,615

tan

tangent

Synopsis

```
#include <math.h>
float tanf (float x)
double tan (double x)
long double tand (long double x)
```

Description

The `tan` function returns the tangent of the argument `x`, where `x` is measured in radians. The library function returns 0 for any input argument that is outside the defined domain.

Algorithm

return = tan(x)

Domain

<code>x = [-6,588,397 ... 6,588,397]</code>	for <code>tanf()</code>
<code>x = [-421,657,424 ... 421,657,424]</code>	for <code>tand()</code>

tanh

hyperbolic tangent

Synopsis

```
#include <math.h>
float tanhf (float x)
double tanh (double x)
long double tanhd (long double x)
```

Description

This function calculates the hyperbolic tangent of number x , where x is measured in radians. If x is outside the domain, this function returns 3.4×10^{38} for a float-type return value and 1.7×10^{308} for a double-type return value.

Algorithm

return = tanh(x)

Domain

$x = [-(\ln(3.4 \times 10^{38}) - \ln(2)) \dots (\ln(3.4 \times 10^{38}) - \ln(2))]$ for `tanhf()`

$x = [-(\ln(1.7 \times 10^{308}) - \ln(2)) \dots (\ln(1.7 \times 10^{308}) - \ln(2))]$ for `tanhd()`

transpm

matrix transpose

Synopsis

```

#include <matrix.h>
void transpmf(a,n,m,c)
const float *a;      /* Pointer to input matrix a[][] */
int n;               /* Number of rows in matrix a[][] */
int m;               /* Number of columns in matrix a[][] */
float *c;             /* Pointer to output matrix c[][] */

```

Description

This function computes the linear algebraic transpose of input matrix `a[][]` and stores the result in `c[][]`. The dimensions of matrix `a` are `n` and `m`. The resulting matrix `c[][]` is of dimensions `m` and `n`.

Algorithm

$$c_{ji} = a_{ij}$$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

twidfft

generate FFT twiddle factors

Synopsis

```
#include <filter.h>
void twidfft(w[], n)
complex_float w[];      /* Twiddle sequence */
int n;                  /* Number of FFT points */
```

Description

Various different versions of the FFT are provided, including `cffft`, `rffft`, `iffft`, `cffft2d`, `rffft2d`, `iffft2d`. The number of points is provided as an argument; when appropriate, the library uses radix-2 or radix-4 implementations.

For efficiency, the twiddle table is calculated once, during initialization, and then provided to the FFT routine as a separate parameter. You must declare the variable and initialize it prior to calling an FFT function. The function `twidfft` is the initialization function.

Several FFTs of different sizes can all be accommodated with the same twiddle factor 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 your FFT uses only half the points of the largest, the stride should be 2 (this takes only every other element).



The twiddle table generated by the `twidfft` function is not compatible with the twiddle table generated by the function `twidfftf`, and must not be used in conjunction with the fast FFT functions `cfftf` and `rfftf`.

Algorithm

This function takes FFT length n as an input parameter and generates the lookup table of complex twiddle coefficients. $3/4$ of one period of sine/cosine is described by $3/4 n$ complex samples in the lookup table.

The samples are generated as follows:

$$twid_re(k) = \cos\left(\frac{2\pi}{n} k\right)$$

$$twid_im(k) = \sin\left(\frac{2\pi}{n} k\right)$$

where $k = \{0, 1, 2, \dots, \frac{3}{4}n - 1\}$

Domain

The n parameter must be either a power of two, or a power of four, and at least 16.

twidfftf

generate FFT twiddle factors for a fast FFT

Synopsis

```
#include <filter.h>
void twidfftf(w[], n)
complex_float w[];      /* Twiddle sequence */
int n;                  /* Number of FFT points */
```

Description

The `twidfftf` function generates complex twiddle factors for the fast FFT functions `cfft` and `rfft`, and stores the coefficients in the vector `w`. The vector `w`, known as the twiddle table, is normally calculated once and is then passed to a fast FFT as a separate argument. The size of the table must be $\frac{1}{2}$ of `n`, the number of points in the FFT.

The same twiddle table may be used to calculate FFTs of different sizes provided that the table was generated for the largest FFT. Each FFT function has a stride parameter that the function uses to stride through the twiddle table. Normally, this stride parameter will be set to 1, but to generate a smaller FFT, the argument should be scaled appropriately. For example, if a twiddle table was generated for an FFT with `N` points, then the same twiddle table may be used to generate a $N/2$ -point FFT provided that the stride parameter is set to 2, or a $N/4$ -point FFT if the parameter is set to 4.



The twiddle table generated by the `twidfftf` function is not compatible with the twiddle table generated by the `twidfft` function.

Algorithm

The function calculates a lookup table of complex twiddle factors. The coefficients generated are:

$$twid_re(k) = \cos\left(\frac{2\pi}{n}k\right)$$

$$twid_im(k) = -\sin\left(\frac{2\pi}{n}k\right)$$

where $k = \{0, 1, 2, \dots, n/2 - 1\}$

Domain

The number of points in the FFT must be a power of 2 and must be at least 32.

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 */
```

Description

This function computes the variance of the input elements contained within input vector *a* and returns the result.

Algorithm

$$c = \frac{n * \sum_{i=0}^{n-1} a_i^2 - (\sum_{i=0}^{n-1} a_i)^2}{n(n-1)}$$

Domain

-3.4 x 10³⁸ to +3.4 x 10³⁸

vecdot

real vector dot product

Synopsis

```
#include <vector.h>
float vecdotf(a,b,n)
const float a[];      /* Input vector a */
const float b[];      /* Input vector b */
int n;                /* Element count */
```

Description

This function computes the dot product of the two input vectors a and b, and returns the scalar result.

Algorithm

$$return = \sum_{i=0}^{n-1} a_i * b_i$$

where $i = \{0,1,2,\dots,n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecsadd

real vector + scalar addition

Synopsis

```
#include <vector.h>
void vecsaddf(a,b,c,n)
const float a[];      /* Input vector a */
float b;               /* Input scalar */
float c[];             /* Output vector */
int n;                /* Element count */
```

Description

This function adds input scalar *b* to each element of input vector *a*. The results are stored in the output vector *c*.

Algorithm

$$c_i = a_i + b$$

where $i = \{0, 1, 2, \dots, n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecsmult

real vector * scalar multiplication

Synopsis

```
#include <vector.h>
void vecsmultf(a,b,c,n)
const float a[];          /* Input vector a */
float b;                   /* Input scalar */
float c[];                 /* Output vector */
int n;                     /* Element count */
```

Description

This function multiplies each element of input vector a by input scalar b. The results are stored in the output vector c.

Algorithm

$$c_i = a_i * b$$

where $i = \{0, 1, 2, \dots, n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecssub

real vector - scalar subtraction

Synopsis

```
#include <vector.h>
void vecssubf(a,b,c,n)
const float a[];      /* Input vector a */
float b;               /* Input scalar */
float c[];             /* Output vector */
int n;                /* Element count */
```

Description

This function subtracts input scalar *b* from each element of input vector *a*. The results are stored in the output vector *c*.

Algorithm

$$c_i = a_i - b$$

where $i = \{0, 1, 2, \dots, n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecvadd

real vector + vector addition

Synopsis

```
#include <vector.h>
void vecvaddf(a,b,c,n)
const float a[];      /* Input vector a */
const float b[];      /* Input scalar b */
float c[];             /* Output vector */
int n;                /* Element count */
```

Description

This function adds two input vectors. The results are stored in the output vector c.

Algorithm

$$c_i = a_i + b_i$$

where $i = \{0,1,2,\dots,n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

vecvmlt

real vector * vector multiplication

Synopsis

```
#include <vector.h>
void vecvmltf(a,b,c,n)
const float a[];      /* Input vector a */
const float b[];      /* Input scalar b */
float c[];             /* Output vector */
int n;                /* Element count */
```

Description

This function multiplies two input vectors a and b. The results are stored in the output vector c.

Algorithm

$$c_i = a_i * b_i$$

where $i = \{0,1,2,\dots,n-1\}$

Domain

$$-3.4 \times 10^{38} \text{ to } +3.4 \times 10^{38}$$

vecvsub

real vector - vector subtraction

Synopsis

```
#include <vector.h>
void vecvsubf(a,b,c,n)
const float a[];          /* Input vector a */
const float b[];          /* Input scalar b */
float c[];                /* Output vector */
int n;                    /* Element count */
```

Description

This function subtracts input vector *b* from input vector *a*. The results are stored in the output vector *c*.

Algorithm

$$c_i = a_i - b_i$$

where $i = \{0, 1, 2, \dots, n-1\}$

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

zero_cross

count zero crossings

Synopsis

```
#include <stats.h>
int zero_crossf(a,n)
const float 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. If all the input values are either positive or zero, or they are all either negative or zero, then the function will return a zero.

Algorithm

The actual algorithm is different from the one shown below because the algorithm needs to handle the case where an element of the array is zero. However, this example should give you an understanding of the algorithm.

```
if (a(i) > 0 && a(i+1) < 0 ) || (a(i) < 0 && a(i+1) > 0)
```

Number of zeros is increased by one.

Domain

-3.4×10^{38} to $+3.4 \times 10^{38}$

I INDEX

Symbols

- #pragma align num, [1-150](#), [2-12](#)
- #pragma all_aligned, [1-154](#), [2-44](#)
- #pragma can_instantiate instance, [1-166](#)
- #pragma const, [1-158](#), [2-39](#)
- #pragma different_banks, [1-155](#), [2-46](#)
- #pragma do_not_instantiate instance, [1-166](#)
- #pragma instantiate instance, [1-166](#)
- #pragma interrupt, [1-151](#)
- #pragma interrupt_reentrant, [1-151](#)
- #pragma linkage_name, [1-163](#)
- #pragma loop_count(min, max, modulo), [1-154](#), [2-42](#)
- #pragma must_iterate(min, max, modulo), [1-155](#)
- #pragma no_alias, [1-156](#), [2-46](#)
- #pragma no_vectorization, [1-154](#), [2-43](#)
- #pragma optimize_as_cmd_line, [1-163](#)
- #pragma optimize_for_space, [1-162](#), [2-42](#)
- #pragma optimize_for_speed, [1-162](#), [2-42](#)
- #pragma optimize_off, [1-162](#), [2-42](#)
- #pragma pure, [1-157](#), [2-40](#)
- #pragma regs_clobbered, [1-158](#), [2-40](#)
- #pragma retain_name, [1-164](#)
- #pragma SIMD_for, [1-156](#)
- #pragma vector_for, [1-155](#), [2-43](#)
- #pragma weak_entry, [1-164](#)
- .IDL files, [1-184](#)
- @ filename (command file) compiler switch, [1-22](#)
- __ADI_LIBEH__ macro, [1-55](#)
- __ADSPTS__ macro, [1-180](#)
- __ADSPTS101__ macro, [1-180](#)
- __ADSPTS201__ macro, [1-180](#)
- __ADSPTS202__ macro, [1-180](#)
- __ADSPTS203__ macro, [1-180](#)
- __ADSPTS20x__ macro, [1-180](#)
- __alignof__ keyword, [1-177](#)
- __ANALOG_EXTENSIONS__ macro, [1-180](#)
- __argv_string variable, [1-197](#)
- __attribute__ keyword, [1-178](#)
- __builtin_prefix, [1-115](#)
- __builtin_aligned function, [1-101](#), [2-6](#), [2-12](#)
- __builtin_aligned function, [2-44](#)
- __builtin_circindex function, [2-35](#), [2-36](#)

INDEX

- `__builtin_circptr` function, [2-35](#), [2-36](#)
 - `__builtin_quad` extended type, [1-59](#), [1-88](#)
 - `__builtin_sysreg_read` built-in function, [1-100](#)
 - `__builtin_sysreg_write` built-in function, [1-100](#)
 - `__cplusplus` macro, [1-180](#)
 - `__DATE__` macro, [1-180](#)
 - `__DOUBLES_ARE_FLOATS__` macro, [1-28](#), [1-180](#)
 - `__ECC__` macro, [1-180](#)
 - `__EDG__` macro, [1-180](#)
 - `__EDG_VERSION__` macro, [1-180](#)
 - `__emuclk` function, [3-111](#)
 - `__EXCEPTIONS` macro, [1-55](#)
 - `__FILE__` macro, [1-180](#)
 - `__GROUPNAME__` macro, [1-47](#)
 - `__HOSTNAME__` macro, [1-47](#)
 - `__IGNORE_IDLE_BUILTINS__` macro, [1-100](#)
 - `__IGNORE_SYSREG_BUILTIN_S__` macro, [1-100](#)
 - `__LINE__` macro, [1-180](#)
 - `__MACHINE__` macro, [1-47](#)
 - `__NO_BUILTIN` macro, [1-180](#)
 - `__NO_BUILTIN` preprocessor macro, [1-36](#)
 - `__NO_BYTE_ADDRESSING__` macro, [1-100](#)
 - `__primIO` label, [3-16](#), [3-17](#)
 - `__REALNAME__` macro, [1-47](#)
 - `__regclass` construct, [1-90](#)
 - `__regclass(unit)` keyword extension, [1-68](#)
 - `__RTTI` macro, [1-56](#)
 - `__SIGNED_CHARS__` macro, [1-46](#), [1-49](#), [1-181](#)
 - `__SILICON_REVISION__` macro, [1-46](#), [1-181](#)
 - `__STDC__` macro, [1-181](#)
 - `__STDC_VERSION__` macro, [1-181](#)
 - `__SYSTEM__` macro, [1-47](#)
 - `__TIME__` macro, [1-181](#)
 - `__TS_BYTE_ADDRESS` macro, [1-25](#), [1-70](#), [1-73](#), [1-181](#)
 - `__USE_RAW_BUILTINS__` macro, [1-100](#)
 - `__USE_RAW_XCORRS__` macro, [1-100](#)
 - `__USERNAME__` macro, [1-47](#)
 - `__VERSION__` macro, [1-181](#)
 - `__WORKAROUNDS_ENABLE_D` macro, [1-181](#)
 - `__XSTAT` registers, [1-129](#)
 - `__YSTAT` registers, [1-129](#)
 - `__ADI_THREADS` macro, [1-48](#)
 - `_primIO()` C function, [3-16](#)
 - `_primio.h` header file, [3-19](#)
 - μ -Law companders, [3-24](#)
 - μ -Law compression, [3-165](#)
 - μ -Law expansion, [3-166](#)
- Numerics**
- 128-bit data types, [1-88](#)

- 16-bit components, [1-133](#)
- 16-bit data types, [1-101](#)
- 2-D convolution, [3-93](#)
- 32-bit data types, [1-57](#)
- 64-bit floating-point arithmetic, [1-59](#)
- 64-bit integer support, [1-87](#)
- A**
 - A (assert) compiler switch, [1-22](#)
 - a_compress (A-law compression)
 - function, [3-42](#)
 - a_expand (A-law expansion)
 - function, [3-43](#)
 - Abridged C++ Library
 - overview, [3-2](#)
 - support, [3-30](#)
 - abs (absolute value) function, [3-44](#)
 - accums pointer, [1-146](#)
 - acos (arc cosine) function, [3-45](#)
 - addbitrev (bit-reversed adder)
 - function, [3-46](#)
 - A-law companders, [3-24](#)
 - algorithm header file, [3-34](#)
 - alias
 - avoiding, [2-14](#)
 - align num pragma, [1-150](#)
 - align-branch-lines compiler switch, [1-23](#)
 - alignment inquiry keyword, [1-177](#)
 - alog10 functions, [3-49](#)
 - alttok (alternative tokens) C++
 - mode compiler switch, [1-23](#)

- anach (enable C++ anachronisms)
 - compiler switch, [1-54](#)
- anachronisms
 - default C++ mode, [1-54](#)
 - disabling in C++ mode, [1-56](#)
- annotations
 - assembly code, [2-51](#)
 - assembly source code position, [2-58](#)
 - loop identification, [2-54](#)
 - modulo scheduling, [2-66](#), [2-67](#)
 - vectorization, [2-64](#)
- ANSI standard, [3-1](#)
 - compilers, [1-29](#)
- anti-log
 - functions, [3-47](#)
- arg (get phase of complex number)
 - functions, [3-51](#)
- argument words, [1-192](#)
- arguments
 - handling, [1-191](#)
 - number, [1-193](#)
- argv/argc support, [1-197](#)
- arithmetic
 - mixed mode operations, [1-170](#)
- arithmetic operators
 - fractional, [1-168](#), [1-169](#)
 - int2x16, [1-104](#)
 - int4x16, [1-109](#)
- array
 - length, [1-94](#)
 - placing in memory, [2-19](#)
 - zero length, [1-175](#)

INDEX

- ASCII string (*see* `atof`, `atoi`, `atol` functions)
 - `asin` (arc sine) function, [3-52](#)
 - `asm`
 - compiler keyword, [1-67](#), [1-75](#)
 - keyword, [1-178](#)
 - statement, [1-176](#), [2-18](#)
 - `asm` compiler keyword (*see also* inline assembly language support keyword (`asm`))
 - `asm()`
 - construct, [1-75](#)
 - operand constraints, [1-83](#)
 - assembly
 - code annotations, [2-51](#)
 - construct
 - reordering and optimization, [1-85](#)
 - template, [1-76](#)
 - with input and output operands, [1-86](#)
 - construct operand, [1-79](#)
 - constructs
 - with multiple instructions, [1-85](#)
 - subroutines, [1-213](#)
 - `assert.h` header file, [3-10](#)
 - `atan` (arc tangent) function, [3-53](#)
 - `atan2` (arc tangent of quotient) function, [3-54](#)
 - `atof` (convert string to double) function, [3-55](#)
 - `atoi` (convert string to integer) function, [3-57](#)
 - `atol` (convert string to long integer) function, [3-58](#)
 - `atold` (convert string to long double) function, [3-59](#)
 - `atoll` (convert string to long long integer) function, [3-61](#)
 - attributes
 - functions, variables and types, [1-178](#)
 - `autocoh` (autocoherence) function, [3-62](#)
 - `autocorr` (autocorrelation) function, [3-63](#)
 - automatic
 - inlining, [1-39](#), [1-61](#), [2-17](#)
 - loop control variables, [2-30](#)
 - variables, [1-88](#)
 - `avg` (mean of two values) functions, [3-64](#)
-
- ## B
- bank
 - qualifier, [2-21](#), [2-46](#)
 - bank type-qualifier, [1-91](#)
 - base 10
 - anti-log functions, [3-49](#)
 - basic arithmetic functions, [3-23](#)
 - binary array search (*see* `bsearch` function)
 - Bit FIFO temporaries register, [1-209](#)
 - bitfields
 - signed, [1-46](#)
 - unsigned, [1-48](#)

- BITS_PER_WORD constant, [3-18](#)
- bitwise logical operators
 - int2x16 values, [1-105](#)
- bool (*see* Boolean type support
 - keywords (bool, true, false))
- Boolean type keywords (bool, true, false), [1-67](#), [1-93](#)
- branch target buffer, [3-6](#)
- bsearch (binary search in sorted array) function, [3-65](#)
- bss (placing data in bsz) compiler switch, [1-24](#)
- build-lib (build library) compiler switch, [1-24](#)
- built-in functions, [1-98](#)
 - builtins.h, [1-99](#)
 - circular buffer, [1-112](#)
 - circular buffer data alignment buffer (DAB), [1-131](#)
 - Communications Logic Unit operation, [1-132](#)
 - data alignment buffer (DAB), [1-129](#)
 - instructions generated by, [1-115](#)
 - math, [1-114](#)
 - optimization guidance, [1-101](#)
 - system, [2-34](#)
- builtins.h header file, [1-99](#), [1-100](#), [1-115](#)
- byte addressing mode, [1-70](#)
 - __TS_BYTE_ADDRESS macro, [1-70](#)
 - byte sizes, [1-70](#)
 - compiler flag for, [1-70](#)

- libraries supporting, [1-73](#)
- pointers, [1-71](#)
- pragma used in, [1-72](#), [1-150](#)
- selecting, [1-3](#), [1-25](#)
- sizeof operator, [1-70](#)

C

- C (comments) compiler switch, [1-24](#)
- c (compile only) compiler switch, [1-24](#)
- C exit routine, [3-5](#)
- C language extensions, [1-67](#)
 - asm keyword, [1-75](#)
 - bool keyword, [1-93](#)
 - C++ style comments, [1-69](#)
 - compound statements, [1-95](#)
 - false keyword, [1-67](#), [1-93](#)
 - indexed initializers, [1-97](#)
 - inline keyword, [1-74](#)
 - non-constant initializers, [1-96](#)
 - segment keyword, [1-67](#)
 - true keyword, [1-67](#), [1-93](#)
- C run-time
 - library files, [3-5](#)
- C run-time library
 - header files, [3-9](#) to [3-22](#)
 - reference, [3-37](#)
- C run-time model, [1-185](#)
- C start-up files
 - ts_hdr_.dlb, [3-5](#)
- C++
 - embedded library, [3-36](#)
 - exception handling, [1-21](#)

INDEX

- exception handling support
 - library, [3-5](#)
 - fractional type support, [1-168](#)
 - header files
 - for C library facilities, [3-33](#)
- c++ (C++ mode) compiler switch, [1-21](#)
- C++ exit routine, [3-6](#)
- C++ mode compiler switches
 - anach (enable C++ anachronisms), [1-54](#)
 - eh (enable exception handling), [1-55](#)
 - no-anach (disable C++ anachronisms), [1-56](#)
 - no-demangle (disable demangler), [1-56](#)
 - no-eh (disable exception handling), [1-56](#)
 - no-rtti (disable run-time type identification), [1-56](#)
 - rtti (enable run-time type identification), [1-56](#)
- C++ programming examples, [1-219](#)
 - complex support, [1-220](#)
 - fract support, [1-219](#)
- C++ run-time
 - library files, [3-5](#)
 - library with exception handling, [3-5](#)
 - support library files, [3-5](#)
 - support library with exception handling, [3-5](#)
- C++ run-time library
 - using linker with, [3-7](#)
- C++ start-up files
 - ts_hdr_cpp_.dll, [3-5](#)
- C/C++ code optimization, [2-2](#)
- C/C++ mode selection switches
 - c++ (C++ mode), [1-21](#)
 - c89, [1-21](#)
- C/C++ run-time
 - environment, [1-185](#)
- c89 (ISO/IEC 9899 1990 standard) compiler switch, [1-21](#)
- cabs (complex absolute value) function, [3-67](#)
- cadd (complex addition) function, [3-68](#)
- callee preserved registers, [1-204](#)
- caller save registers, [1-204](#)
- calling
 - assembly language subroutine, [1-214](#)
 - C/C++ library functions, [3-3](#)
- calloc function, [1-202](#)
- cartesian (Cartesian to polar) functions, [3-69](#)
- cassert header, [3-33](#)
- cctype header, [3-33](#)
- cdiv (complex division) function, [3-71](#)
- ceil (ceiling) function, [3-72](#)
- cerno header, [3-33](#)
- cexp (complex exponential) function, [3-73](#)
- cfft (N point complex input FFT) function, [3-74](#)

- cfft magnitude, [3-77](#)
- cfft_mag function, [3-77](#)
- cfft2d (NxN point 2-d complex input FFT) function, [3-79](#)
- cfft (fast N point complex input FFT) function, [3-81](#)
- cfftN (N-point complex input fast Fourier transform) functions, [3-179](#), [3-185](#)
- cfloat header, [3-33](#)
- char-size-8|32 compiler switch, [1-25](#)
- char-size-any compiler switch, [1-24](#)
- circular buffer
 - built-in functions, [1-112](#)
 - DAB intrinsics, [1-131](#)
 - in DSP code, [2-35](#)
 - increment of index, [1-112](#)
 - increment of pointer, [1-113](#)
 - operation on array, [1-113](#)
 - switch setting, [1-30](#)
- climits header, [3-33](#)
- clip (clip) functions, [3-83](#)
- CLIP instruction, [3-114](#)
- clobber string, [1-83](#)
 - specifiers, [1-83](#)
- clobbered
 - register sets, [1-160](#)
 - registers, [1-158](#), [1-159](#)
- locale header, [3-33](#)
- cmath header, [3-33](#)
- cmatmadd (complex matrix + matrix addition) function, [3-84](#)
- cmatmmlt (complex matrix * matrix multiplication) function, [3-85](#), [3-90](#)
- cmatmsub (complex matrix - matrix subtraction) function, [3-86](#)
- cmatsadd (complex matrix + scalar addition) function, [3-87](#)
- cmatsmult (complex matrix * scalar multiplication) function, [3-88](#)
- cmatssub (complex matrix - scalar subtraction) function, [3-89](#)
- cmlt (complex multiply) function, [3-90](#)
- code optimization
 - enabling, [1-39](#)
- code sequences, [1-195](#)
- command-line interface, [1-4](#) to [1-56](#)
- comparison operators, [1-105](#)
 - int2x16 values, [1-105](#)
- compiler
 - C extensions, [1-67](#)
 - code optimization, [2-2](#)
 - optimizer, [2-4](#)
 - prelinker, [1-63](#)
 - specifying functional options, [1-9](#)
- compiler errors
 - maximum, [1-51](#)
- complex
 - absolute value function, [3-67](#)
 - addition function, [3-68](#)
 - conjugate function, [3-91](#)
 - division function, [3-71](#)
 - exponential function, [3-73](#)

INDEX

- header file, [3-30](#)
- matrix - matrix subtraction
 - function, [3-86](#)
- matrix - scalar subtraction
 - function, [3-89](#)
- matrix * matrix multiplication
 - function, [3-85](#)
- matrix + matrix addition function, [3-84](#)
- matrix + scalar addition function, [3-87](#)
- multiply function, [3-90](#)
- number support, [1-220](#)
- subtraction function, [3-101](#)
- vector - scalar subtraction
 - function, [3-105](#)
- vector - vector subtraction
 - function, [3-108](#)
- vector * scalar multiplication
 - function, [3-104](#)
- vector * vector multiplication
 - function, [3-107](#)
- vector + scalar addition function, [3-103](#)
- vector + vector addition function, [3-106](#)
- vector dot product function, [3-102](#)
- complex matrix * scalar
 - multiplication function, [3-88](#)
- complex.h header file, [3-23](#)
- compound statements
 - as macros, [1-182](#)
 - within expressions, [1-95](#)
- Compute Block registers
 - (x & y) general, [1-207](#)
 - ALU summation, [1-209](#)
 - X MAC, [1-208](#)
 - Y MAC, [1-209](#)
- conditional code
 - in loops, [2-27](#)
- conditional expressions
 - with missing operands, [1-174](#)
- conj (complex conjugate) function, [3-91](#)
- const
 - keyword, [2-31](#)
 - pointers, [1-25](#)
 - qualifier, [2-31](#)
- const-read-write compiler switch, [1-25](#)
- const-read-write flag, [2-32](#)
- constructors
 - int2x16 values, [1-103](#)
 - int2x32 values, [1-111](#)
 - int4x16 values, [1-107](#)
- constructs
 - flow control, [1-87](#)
 - from polar coordinates (polar function), [3-168](#)
 - operand
 - assembly, [1-79](#)
 - reordering and optimization, [1-85](#)
 - with input and output operands, [1-86](#)
 - with multiple instructions, [1-85](#)
- conv2d (2-d convolution) function, [3-93](#)

- convolution transformations, [3-24](#)
- convolve (convolution) function, [3-92](#)
- copysign (copysign) function, [3-94](#)
- cos (cosine) function, [3-95](#)
- cosh (hyperbolic cosine) function, [3-96](#)
- cot (cotangent) function, [3-97](#)
- count zero crossings function, [3-224](#)
- count_ones (count one bits in word) function, [3-98](#)
- crosscoh (cross-coherence) function, [3-99](#)
- crosscorr (cross-correlation) function, [3-100](#)
- csetjmp header, [3-34](#)
- csignal header, [3-34](#)
- cstdarg header, [3-34](#)
- cstddef header, [3-34](#)
- cstdio header, [3-34](#)
- cstring header, [3-34](#)
- csub (complex subtraction) function, [3-101](#)
- ctype.h header file, [3-10](#)
- custom processors, [1-43](#)
- cvecdot (complex vector dot product) function, [3-102](#)
- cvecsadd (complex vector + scalar addition) function, [3-103](#)
- cvecsmult (complex vector * scalar multiplication) function, [3-104](#)
- cvecssub (complex vector - scalar subtraction) function, [3-105](#)

- cvecvadd (complex vector + vector addition) function, [3-106](#)
- cvecvmult (complex vector * vector multiplication) function, [3-107](#)
- cvecvsub (complex vector - vector subtraction) function, [3-108](#)

D

- D (define macro) compiler switch, [1-25](#), [1-48](#)
- DAB built-in functions, [1-129](#)
- data
 - alignment pragmas, [1-150](#)
 - alignment registers, [1-209](#)
 - buffers
 - quad-word-aligned, [2-11](#)
 - fetching with 32-bit loads, [2-11](#)
 - packing, [3-18](#)
 - word alignment, [2-12](#)
- data object
 - aggregate, [1-59](#)
- data types
 - 128-bit quad word, [1-88](#)
 - 16-bit, [1-101](#)
 - 32-bit, [1-57](#), [1-111](#)
 - 64-bit integer, [1-58](#), [1-87](#)
 - alignment, [1-59](#)
 - default bit sizes, [1-57](#)
 - floating-point, [1-58](#)
 - fract, [1-168](#)
 - integer, [1-58](#)
 - scalar, [2-7](#)
- debugging information, [1-45](#)
 - for header file, [1-26](#)

INDEX

- debug-types compiler switch, [1-26](#)
- declarations
 - mixed with code, [1-177](#)
- decrements in expressions, [1-167](#)
- dedicated registers, [1-204](#), [1-214](#)
- default
 - target processor, [1-43](#)
- default-branch-(np|p) compiler switch, [1-26](#)
- 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](#)
- define
 - interrupt handling, [3-148](#)
 - signal handling, [3-190](#)
- demangler
 - disabling, [1-56](#)
- deque header file, [3-34](#)
- device
 - driver, [1-198](#)
 - identifiers, [1-198](#)
- DeviceID field, [1-198](#)
- div (division) function, [3-109](#)
- division (*see* div, ldiv functions)
- dm memory keyword, [1-67](#), [1-88](#)
- double type formats, [1-27](#)
- double-size-32|64 switch, [1-27](#)
- double-size-any switch, [1-27](#)
- dry (verbose dry-run) compiler switch, [1-28](#)
- dryrun compiler switch, [1-28](#)

- DSP header files, [3-22](#)
 - complex.h, [3-23](#)
 - filter.h header file, [3-24](#)
 - libsim.h, [3-25](#)
 - matrix.h, [3-26](#)
 - stats.h, [3-27](#)
 - vector.h, [3-28](#)
 - window.h, [3-29](#)
- DSP run-time
 - library files, [3-5](#)
- dual memory support keywords
 - (pm dm), [1-67](#)

E

- E (stop after preprocessing)
 - compiler switch, [1-28](#)
- ED (run after preprocessing to file)
 - compiler switch, [1-29](#)
- EE (run after preprocessing)
 - compiler switch, [1-29](#)
- eh (enable exception handling)
 - compiler switch, [1-55](#)
- elfar (archive library), [1-2](#), [1-24](#)
- embedded C++ library
 - header files, [3-30](#)
- embedded standard template
 - library, [3-2](#)
 - header files, [3-34](#)
- emuclk (get simulator cycle count)
 - function, [3-111](#)
- emulated arithmetic
 - avoiding, [2-8](#)
- errno.h header file, [3-10](#)
- escape character, [1-177](#)

- exception handler
 - disabling, [1-56](#)
 - enabling, [1-55](#)
- exception handling, [1-21](#)
- exception header file, [3-31](#)
- exp (exponential) function, [3-110](#)
- exponentiation, [3-47](#), [3-49](#)
- extensions
 - using compiler language, [1-3](#)
- extractors
 - 2x16 from a 4x16 value, [1-108](#)
 - and expanders
 - int2x16 values, [1-103](#)
 - int2x32 values, [1-112](#)
- extra-keywords (enable short-form keywords) compiler switch, [1-29](#)
- EZ-KIT Lite, [1-198](#)

F

- fabs (float absolute value) functions, [3-112](#)
- false (*see* Boolean type support keywords (bool, true, false))
- Fast Fourier Transforms (FFT), [3-24](#)
- fast N point complex input FFT (cfft) function, [3-81](#)
- fast N point real input FFT(rfft) function, [3-183](#)
- fast radix-2 algorithm, [3-24](#), [3-177](#)
- favg (mean of two values) functions, [3-113](#)
- fclip functions, [3-114](#)

- FFT
 - function versions, [3-24](#)
 - twiddle factors, [3-212](#)
 - twiddle factors for a fast FFT, [3-214](#)
- file
 - annotation position, [2-58](#)
 - extension, [1-5](#), [1-7](#)
 - I/O support, [1-197](#)
 - names, [1-22](#)
 - searches, [1-7](#)
- filter.h, [3-24](#)
- filters
 - signal processing, [3-24](#)
- finite impulse response filter., [3-115](#)
- FIR (finite impulse response filter) function, [3-115](#)
- fir_decima (FIR decimation filter) function, [3-117](#)
- fir_init macro, [3-120](#)
- fir_interp (FIR interpolation filter) function, [3-119](#)
- flags (command line input) compiler switch, [1-29](#)
- float.h header file, [3-10](#)
- floating-point
 - arithmetics, [1-58](#)
 - hexadecimal constants, [1-174](#)
- floor (floor) function, [3-124](#)
- flow control operations, [1-87](#)
- FLT_ROUNDS macro, [3-10](#)
- fmax functions, [3-125](#)
- fmin functions, [3-126](#)

INDEX

fmod (floating-point modulus)
 function, [3-127](#)
-force-circbuf (circular buffer)
 compiler switch, [1-30](#)
-force-circbuf switch, [2-35](#)
-fp-associative (floating-point
 associative operation) compiler
 switch, [1-30](#)
fprintf function, [3-17](#)
fract
 header file, [3-31](#)
 support, [1-219](#)
fractional
 (fixed-point) arithmetic, [1-168](#)
 arithmetic operators, [1-169](#)
 mixed mode operations, [1-170](#)
 values, [1-168](#)
frame pointer, [1-187](#)
free function, [1-202](#)
frexp (separate fraction and
 exponent) function, [3-128](#)
fstream header file, [3-31](#)
fstream.h header file, [3-36](#)
-full-version (display versions)
 compiler switch, [1-30](#)
function
 inlining, [2-17](#)
function calls
 avoiding loops in, [2-28](#)
functional header file, [3-34](#)
functions
 primitive I/O, [3-14](#)

G

-g (generate debug information)
 compiler switch, [1-30](#)
GCC compatibility extensions,
 [1-171](#)
gen_bartlett (generate bartlett
 window) function, [3-129](#)
gen_blackman (generate blackman
 window) function, [3-131](#)
gen_gaussian (generate gaussian
 window) function, [3-132](#)
gen_hamming (generate hamming
 window) function, [3-133](#)
gen_hanning (generate hanning
 window) function, [3-134](#)
gen_harris (gen_harris window)
 function, [3-135](#)
gen_kaiser (generate kaiser window)
 function, [3-136](#)
gen_rectangular (generate
 rectangular window) function,
 [3-137](#)
gen_triangle (generate triangle
 window) function, [3-138](#)
gen_vohann (generate von hann
 window) function, [3-140](#)
general optimization pragmas,
 [1-162](#)
get phase of a complex number (arg
 function), [3-51](#)
globvar global variable, [2-30](#)
GNU C compiler, [1-171](#)

H

-H (list headers) compiler switch,
1-31

hash_map header file, 3-34

hash_set header file, 3-35

header files, 1-181

 C run-time library, 3-9 to 3-22

 assert.h, 3-10

 ctype.h, 3-10

 errno.h, 3-10

 float.h, 3-10

 iso646.h, 3-11

 limits.h, 3-11

 locale.h, 3-12

 math.h, 3-12

 setjmp.h, 3-13

 signal.h, 3-13

 stdarg.h, 3-14

 stddef.h, 3-14

 stdio.h, 3-14

 stdlib.h, 3-21

 string.h, 3-22

 time.h, 3-22

 C++ access to C facilities

 cassert, 3-33

 cctype, 3-33

 cerrno, 3-33

 cfloat, 3-33

 climits, 3-33

 clocale, 3-33

 cmath, 3-33

 csetjmp, 3-34

 csignal, 3-34

 cstdarg, 3-34

 cstddef, 3-34

 cstdio, 3-34

 cstdlib, 3-34

 cstring, 3-34

 embedded C++ library

 complex, 3-30

 exception, 3-31

 fract, 3-31

 fstream, 3-31

 iomanip, 3-31

 ios, 3-31

 iosfwd, 3-31

 iostream, 3-31

 new, 3-32

 ostream, 3-32

 sstream, 3-32

 stdexcept, 3-32

 stream, 3-32

 streambuf, 3-32

 string, 3-32

 strstream, 3-33

 embedded standard template

 library, 3-34

 algorithm, 3-34

 deque, 3-34

 fstreams.h, 3-36

 functional, 3-34

 hash_map, 3-34

 hash_set, 3-35

 iomanip.h, 3-36

 iterator, 3-35

 list, 3-35

 map, 3-35

 memory, 3-35

INDEX

- new.h, [3-36](#)
- numeric, [3-35](#)
- ostream.h, [3-36](#)
- queue, [3-35](#)
- set, [3-35](#)
- stack, [3-35](#)
- utility, [3-36](#)
- vector, [3-36](#)
- library, [3-8](#)
- system, [1-181](#)
- user, [1-181](#)
- header files (standard)
 - stdio.h, [1-197](#)
- heap functions
 - alternate, [1-202](#)
 - alternative interface, [1-202](#)
 - calloc, [1-200](#)
 - free, [1-200](#)
 - initializing, [1-202](#)
 - malloc, [1-200](#)
 - realloc, [1-200](#)
 - standard, [1-200](#)
 - standard interface, [1-202](#)
 - using multiple, [1-200](#)
- heap identifier, [1-201](#)
- help (command line help) compiler
 - switch, [1-31](#)
- hexadecimal floating-point
 - constants, [1-174](#)
- HH (list headers and compile)
 - compiler switch, [1-31](#)
- histogram function, [3-141](#)

I

- I (start include directory) compiler
 - switch, [1-31](#), [1-32](#), [1-38](#)
- I/O
 - primitives, [1-197](#), [1-198](#)
 - support for new devices, [1-198](#)
- I/O primitives, [3-17](#)
 - data packing, [3-18](#)
 - data structure, [3-19](#)
- I/O run-time library, [3-5](#)
- iALU (j & k) Special registers:
 - Circular Buffering –B (base) & L (length), [1-208](#)
- iALU (j and k) general registers,
 - [1-206](#)
- IDDE_ARGS macro, [1-197](#)
- ifft (N point inverse FFT) function,
 - [3-142](#)
- ifft2d (NxN point 2-d inverse input FFT) function, [3-144](#)
- iir (infinite impulse response filter)
 - function, [3-146](#)
- iir_init macro, [3-146](#)
- include (include file) compiler
 - switch, [1-32](#)
- include directives, [1-184](#)
- increments in expressions, [1-167](#)
- indexed
 - array, [2-16](#)
- indexed initializer support, [1-96](#)
- induction variables, [2-26](#)
- initiation interval, [2-66](#)
- inline
 - asm statements, [2-18](#)

- avoiding code, [2-38](#)
- keyword, [2-17](#), [2-38](#)
- inline assembly language support
 - keyword (asm)
 - construct
 - I/O operands, [1-86](#)
 - optimization, [1-85](#)
 - template, [1-76](#)
 - template operands, [1-80](#)
 - with multiple instructions, [1-85](#)
 - construct template, [1-76](#)
- inline keyword
 - (asm), [1-75](#)
 - (inline), [1-67](#), [1-74](#)
- inlining
 - automatic, [2-17](#)
 - file position, [2-58](#)
 - function, [2-17](#)
- inner loop, [2-27](#)
- instantiation
 - template functions, [1-165](#)
- instructions
 - ACS, [1-139](#)
 - addition, [1-116](#)
 - ALU miscellaneous, [1-119](#)
 - bit manipulation, [1-123](#)
 - composition, [1-127](#)
 - conversion, [1-119](#)
 - decomposition, [1-127](#)
 - DESPREAD, [1-143](#)
 - executed by Communications
 - Logic Unit (CLU), [1-132](#)
 - floating-point operation, [1-126](#)
 - generated by built-in functions,
 - [1-115](#)
 - MAX_ADD, [1-133](#)
 - MAX_SUB, [1-133](#)
 - memory allocation, [1-127](#)
 - miscellaneous, [1-127](#)
 - multiplier, [1-125](#)
 - PERMUTE, [1-138](#)
 - shifter, [1-122](#)
 - subtraction, [1-116](#)
 - system register access, [1-128](#)
 - TMAX, [1-133](#)
 - TMAX_ADD, [1-133](#)
 - TMAX_SUB, [1-133](#)
 - XCORRS, [1-145](#)
- int2x32 data type, [1-111](#)
- integer data types, [1-58](#)
- interfacing C/C++ and assembly
 - (see mixed C/C++/assembly programming)
- interpolation factor, [3-119](#)
- interprocedural analysis, [1-33](#), [1-62](#), [2-6](#)
- interrupt
 - dispatcher, [3-149](#), [3-191](#)
 - function, [3-148](#)
 - handler, [1-151](#)
 - handler pragmas, [1-151](#)
- interrupt handler, [1-152](#)
- interrupt handlers
 - non-reentrant, [3-150](#)
- interrupt_reentrant pragma, [1-151](#)
- interruptf function, [3-148](#), [3-149](#)
- interruptfnr function, [3-150](#)
- interruptnr function, [3-150](#)

INDEX

interrupts function, [3-148](#), [3-150](#)
interruptsnr function, [3-150](#)
intrinsic, *see* built-in functions, [1-98](#)
Inverse Fast Fourier Transform (IFFT), [3-142](#)
iomanip header file, [3-31](#)
iomanip.h header file, [3-36](#)
ios header file, [3-31](#)
iosfwd header, [3-31](#)
iostream header file, [3-31](#)
iostream.h header file, [3-36](#)
IPA, [1-62](#)
-ipa (interprocedural analysis) compiler switch, [1-33](#), [2-6](#)
iso646.h (Boolean operator) header file, [3-11](#)
istream header file, [3-32](#)
iterator header file, [3-35](#)

K

keywords (compiler) (*see* VisualDSP++ compiler C/C++ language extensions)

L

-L (library search directory) compiler switch, [1-33](#)
-l (link library) compiler switch, [1-33](#), [1-38](#)
language extensions (compiler) (*see* VisualDSP++ compiler C/C++ language extensions)
LDBL_MAX value, [3-60](#)

leaf
 procedure, [1-195](#)
libc_.dlb library files, [3-5](#)
libcpp_.dlb files, [3-5](#)
libcpprt_.dlb library files, [3-5](#)
libdsp_.dlb library files, [3-5](#)
libio_.dlb library files, [3-5](#)
library
 C run-time, [3-5](#)
 functions, listed, [3-37](#)
 header files, working with, [3-8](#)
 source code, working with, [3-7](#)
libsim print routines, [3-25](#)
libsim.dlb library, [3-5](#), [3-17](#)
libsim.h header file, [3-25](#)
libx_.dlb library files, [3-5](#)
limits.h header file, [3-11](#)
line breaks
 in string literals, [1-176](#)
linking
 pragmas for, [1-163](#)
list header file, [3-35](#)
local
 variables and temporaries, [1-188](#)
locale.h header file, [3-12](#)
log (natural logarithm) function, [3-152](#)
log10 (base 10 logarithm) function, [3-153](#)
logical operators
 bitwise, [1-105](#)
long
 latencies, [2-32](#)
loop

- annotations, [2-67](#)
- control variables, [2-30](#)
- counters, [1-209](#)
- cycle count, [2-56](#)
- exit test, [2-30](#)
- flattening, [2-63](#)
- identification annotation, [2-54](#),
[2-55](#)
- iteration count, [2-42](#)
- optimization, [1-154](#), [2-42](#)
- parallel processing, [1-155](#)
- resource usage, [2-56](#)
- rotation by hand., [2-25](#)
- scheduled, [2-66](#)
- short, [2-23](#)
- trip count, [2-60](#)
- unrolling, [2-23](#)
- vectorization, [2-43](#)
- loop-carried dependency, [2-24](#),
[2-25](#)
- lvalue
 - GCC generalized, [1-173](#)
 - generalized, [1-173](#)

M

- M (make only) compiler switch,
[1-33](#)

macros

- `__GROUPNAME__`, [1-47](#)
- `__HOSTNAME__`, [1-47](#)
- `__MACHINE__`, [1-47](#)
- `__REALNAME__`, [1-47](#)
- `__RTTI`, [1-56](#)
- `__SIGNED_CHARS__`, [1-49](#)

- `__SYSTEM__`, [1-47](#)
- `__TS_BYTE_ADDRESS`, [1-25](#),
[1-70](#)
- `__USERNAME__`, [1-47](#)
- `_ADI_THREADS`, [1-48](#)
- compound statements as, [1-182](#)
- defining, [1-25](#)
- `FLT_ROUNDS`, [3-10](#)
- `IDDE_ARGS`, [1-197](#)
- preprocessor, [1-179](#)
- undefining, [1-48](#)
- variable argument, [1-175](#)
- writing preprocessor, [1-182](#)
- malloc function, [1-202](#)
- map (generate a memory map)
 - compiler switch, [1-35](#)
- map header file, [3-35](#)
- math intrinsics, [1-114](#)
- math.h header file, [3-12](#)
- mathematical functions
 - floating point, [3-12](#)
- matinv (matrix inversion) function,
[3-154](#)
- matmadd (real matrix matrix
addition) function, [3-155](#)
- matmmlt (real matrix matrix
multiplication) function, [3-156](#)
- matmsub (real matrix matrix
subtraction) function, [3-157](#)
- matrix functions, [3-26](#)
- matrix inversion, [3-154](#)
- matrix transpose, [3-211](#)
- matrix.h header file, [3-26](#)

INDEX

- matsadd (real matrix scalar addition) function, [3-158](#)
- matxslt (real matrix scalar multiplication) function, [3-159](#)
- matssub (real matrix scalar subtraction) function, [3-160](#)
- max (maximum) functions, [3-161](#)
- MAX instruction, [3-161](#)
- maximum performance., [2-37](#)
- MD (make and compile) compiler switch, [1-34](#)
- mean (mean) function, [3-162](#)
- mem (enable memory initialization) compiler switch, [1-35](#)
- mem21k initializer
 - disabling, [1-37](#)
- meminit_.dlb files, [3-6](#)
- memory
 - data placement in, [2-19](#)
 - header file, [3-35](#)
 - initialization, [1-35](#)
- memory initialization support
 - meminit_ files, [3-6](#)
- memory initializer
 - disabling, [1-37](#)
 - enabling, [1-35](#)
- memory support keywords
 - (pm dm), [1-88](#)
- min (minimum) functions, [3-163](#)
- MIN instruction, [3-163](#)
- minimum code size, [2-37](#)
- missing operands
 - in conditional expressions, [1-174](#)
- mixed C/assembly programming
 - asm() constructs, [1-76](#)
- mixed C/assembly naming conventions, [1-217](#)
- mixed C/assembly programming
 - asm() constructs, [1-75](#), [1-76](#), [1-80](#), [1-85](#), [1-86](#)
- mixed C/C++/assembly programming, [1-185](#)
- mixed C/C++/assembly reference, [1-216](#)
- MM (generate make rules and compile) compiler switch, [1-34](#)
- Mo (processor output file) compiler switch, [1-34](#)
- modf (separate integral and fractional parts) function, [3-164](#)
- modulo scheduled loop, [2-68](#)
- modulo scheduling, [2-66](#)
 - annotations, [2-67](#)
- modulo variable expansion unroll, [2-67](#)
- MQ compiler switch, [1-34](#)
- Mt filename (output make rule) compiler switch, [1-34](#)
- mu_compress (μ -law compression) function, [3-165](#)
- mu_expand (μ -law expansion) function, [3-166](#)
- multidimensional arrays, [1-95](#)
- multi-line asm() C program constructs, [1-85](#)

multiple instructions
constructs with, [1-85](#)

N

N point complex input FFT (cfft)
function, [3-74](#)

N point inverse FFT (ifft) function,
[3-142](#)

N point real input FFT (rfft)
function, [3-176](#)

new devices

I/O support, [1-198](#)

new header file, [3-32](#)

new.h header file, [3-36](#)

-no-align-branch-lines compiler
common switch, [1-35](#)

-no-alttok (disable tokens) compiler
common switch, [1-35](#)

-no-anach (disable C++
anachronisms) compiler switch,
[1-56](#)

-no-annotate (disable assembly
annotations) compiler common
switch, [1-35](#)

-no-bss compiler common switch,
[1-36](#)

-no-builtin (no built-in functions)
compiler switch, [1-36](#)

-no-circbuf (no circular buffer)
compiler switch, [1-36](#)

-no-const-strings compiler switch,
[1-36](#)

-no-def (disable definitions)
compiler switch, [1-37](#)

-no-demangle compiler switch, [1-56](#)

-no-eh (disable exception handling)
C++ mode compiler switch,
[1-56](#)

-no-extra-keywords (disable
short-form keywords) compiler
switch, [1-37](#)

-no-fp-associative compiler switch,
[1-37](#)

-no-mem (disable memory
initialization) compiler switch,
[1-37](#)

non-constant initializer support
(compiler), [1-96](#)

non-unit stride
avoiding, [2-29](#)

norm (normalization) functions,
[3-167](#)

-no-rtti (disable run-time type
identification) C++ mode
compiler switch, [1-56](#)

-no-saturation (no faster operations)
compiler switch, [1-37](#)

-no-std-ass (disable standard
assertions) compiler switch,
[1-38](#)

-no-std-def (disable standard
definitions) compiler switch,
[1-38](#)

-no-std-inc (disable standard
include search) compiler switch,
[1-38](#)

-no-std-lib (disable standard library
search) compiler switch, [1-38](#)

INDEX

- nothreads (disable thread-safe build) compiler switch, [1-38](#)
- numeric header file, [3-35](#)
- NxN point 2-D complex input FFT (cfft2d) function, [3-79](#)
- NxN point 2-d inverse input FFT (ifft2d) function, [3-144](#)
- NxN point 2-D real input FFT (rfft2d) function, [3-181](#)

O

- O (enable optimization) compiler switch, [1-39](#)
- o (output) compiler switch, [1-40](#)
- Oa (automatic function inlining) compiler switch, [1-39](#)
- operand constraints, [1-81](#)
- operators
 - comparison, [1-105](#)
- optimization
 - code size, [2-37](#)
 - compiler code, [2-4](#)
 - default, [1-60](#)
 - enabling, [1-39](#)
 - procedural, [1-60](#)
 - switches, [2-47](#)
 - turning on, [1-63](#)
- optimizer
 - compiler, [2-4](#)
- optimizing
 - asm() C program constructs, [1-85](#)
 - for code size, [2-37](#)
 - for speed, [2-37](#)

- Os (optimize for size) compiler switch, [1-39](#)
- ostream header file, [3-32](#)
- outer loop, [2-27](#)
- Ov (optimize for speed vs. size) compiler switch, [1-40](#)
- Ov num (optimize for speed versus size) compiler switch, [1-40](#)

P

- P (omit line numbers) compiler switch, [1-40](#)
- packed 16-bit integer support, [1-103](#)
- path-install (installation location) compiler switch, [1-41](#)
- path-output (non-temporary files location) compiler switch, [1-41](#)
- path-temp (temporary files location) compiler switch, [1-41](#)
- path-tool (tool location) compiler switch, [1-40](#)
- pch (precompiled header) compiler switch, [1-41](#)
- pchdir (locate PCHRepository) compiler switch, [1-42](#)
- pedantic (ANSI standard warnings) compiler switch, [1-42](#)
- pedantic-errors (ANSI C errors) compiler switch, [1-42](#)
- peeled iterations, [2-61](#)
- peeling amount, [2-61](#)

- pguide (profile-guided optimization) compiler switch, [1-42](#)
- Pipeline Viewer, [2-32](#)
- placement support keyword (segment), [1-67](#), [1-92](#)
- pm memory keyword, [1-67](#), [1-88](#)
- pointer
 - arithmetic action on, [1-176](#)
 - byte-addressing mode, [1-71](#)
 - incrementing, [2-16](#)
- pointer class support keyword (restrict), [1-67](#), [1-93](#)
- polar (construct from polar coordinates) functions, [3-168](#)
- polyphase interpolation filter, [3-119](#)
- pow (raise to a power) function, [3-169](#)
- PP (omit line numbers and compile) compiler switch, [1-40](#)
- pplist (preprocessor listing) compiler switch, [1-42](#)
- pragmas, [1-148](#)
 - align num, [1-150](#), [1-154](#)
 - can_instantiate instance, [1-166](#)
 - const, [1-158](#)
 - data alignment, [1-150](#)
 - different_banks, [1-155](#)
 - do_not_instantiate instance, [1-166](#)
 - function side-effect, [1-157](#)
 - instantiate instance, [1-166](#)
 - interrupt, [1-151](#)
 - linkage_name, [1-163](#)
 - linking, [1-163](#)
 - linking control, [1-163](#)
 - loop optimization, [1-154](#), [2-42](#)
 - loop_count(min, max, modulo), [1-154](#)
 - no_alias, [1-156](#)
 - no_vectorization, [1-154](#)
 - optimize_for_space, [1-162](#)
 - optimize_off, [1-162](#)
 - pure, [1-157](#)
 - regs_clobbered string, [1-158](#)
 - retain_name, [1-164](#)
 - template instantiation, [1-165](#)
 - vector_for, [1-155](#)
 - weak_entry, [1-164](#)
- precompiled header, [1-41](#)
- predefined macros, [1-179](#)
 - __ADSPTS__, [1-180](#)
 - __ADSPTS101__, [1-180](#)
 - __ADSPTS201__, [1-180](#)
 - __ADSPTS202__, [1-180](#)
 - __ADSPTS203__, [1-180](#)
 - __ADSPTS20x__, [1-180](#)
 - __ANALOG_EXTENSIONS__, [1-180](#)
 - __cplusplus, [1-180](#)
 - __DATE__, [1-180](#)
 - __DOUBLES_ARE_FLOATS__, [1-180](#)
 - __ECC__, [1-180](#)
 - __EDG__, [1-180](#)
 - __EDG_VERSION__, [1-180](#)
 - __FILE__, [1-180](#)
 - __LINE__, [1-180](#)

INDEX

`__NO_BUILTIN`, [1-180](#)
`__SIGNED_CHARS__`, [1-181](#)
`__SILICON_REVISION__`,
 [1-181](#)
`__STDC__`, [1-181](#)
`__STDC_VERSION__`, [1-181](#)
`__TIME__`, [1-181](#)
`__TS_BYTE_ADDRESS`, [1-181](#)
`__VERSION__`, [1-181](#)
`__WORKAROUNDS_ENABLED`, [1-181](#)
prelinker, [1-63](#)
preprocessing
 IDL files, [1-184](#)
preprocessor macros, [1-179](#)
primitive I/O functions, [3-14](#), [3-18](#)
print routines, [3-25](#)
printf functions, [3-15](#), [3-17](#)
printf routine, [3-25](#)
-proc (target processor) compiler
 switch, [1-43](#)
procedural optimizations, [1-60](#)
procedure call, [1-190](#)
profile-guided optimization, [1-61](#),
 [2-5](#), [2-38](#)
 enabling, [1-42](#)
program termination, [3-15](#)

Q

qsort (quicksort) function, [3-170](#)
quad-word
 boundary, [2-11](#), [2-13](#)
 data type, [1-88](#)
quad-word-aligned address, [2-11](#)

queue header file, [3-35](#)

R

-R- (disable source path) compiler
 switch, [1-44](#)
-R directory (add source directory)
 compiler switch, [1-44](#)
raise (raise a signal) function, [3-172](#)
raise to a power function, [3-169](#)
rand (random number generator)
 function, [3-175](#)
real vector - scalar subtraction
 function, [3-220](#)
real vector - vector subtraction
 function, [3-223](#)
real vector * scalar multiplication
 function, [3-219](#)
real vector * vector multiplication
 function, [3-222](#)
real vector + scalar addition
 function, [3-218](#)
real vector + vector addition
 function, [3-221](#)
real vector dot product function,
 [3-217](#)
realloc function, [1-202](#)
reciprocal square root (rsqrt)
 function, [3-188](#)
reductions, [2-24](#)
registers, [1-204](#)
 ADSP-TS101, [1-205](#)
 ADSP-TS201, [1-205](#)
 Bit FIFO temporaries, [1-209](#)
 callee preserved, [1-204](#)

- classification, [1-204](#)
 - clobbered, [1-158](#)
 - Compute Block (X & Y) General, [1-207](#)
 - Compute Block ALU
 - Summation, [1-209](#)
 - Compute Block X MAC, [1-208](#)
 - Compute Block Y MAC, [1-209](#)
 - Data Alignment, [1-209](#)
 - enhanced communications, [1-210](#), [1-211](#)
 - for asm() constructs, [1-80](#)
 - iALU (j & k) special registers:
 - Circular Buffering –B (base) & L (length), [1-208](#)
 - iALU (j and k) General, [1-206](#)
 - loop counter, [1-209](#)
 - qualifier, __regclass, [1-90](#)
 - Return Address, [1-210](#)
 - scratch, [1-204](#)
 - Static Condition Flags, [1-209](#)
 - regs_clobbered string, [1-159](#)
 - reordering asm() C program
 - constructs, [1-85](#)
 - restrict
 - keyword, [2-31](#)
 - qualifier, [2-31](#)
 - restricted pointer, [2-31](#)
 - retain_name pragma, [1-164](#)
 - Return Address register, [1-210](#)
 - return values, [1-193](#)
 - rfft (N point real input FFT)
 - function, [3-176](#)
 - rfft magnitude, [3-179](#)
 - rfft_mag function, [3-179](#)
 - rfft2d (NxN point 2-D real input FFT) function, [3-181](#)
 - rfftf (fast N point real input FFT) function, [3-183](#)
 - rfftf magnitude, [3-185](#)
 - rfftf_mag function, [3-185](#)
 - rms (root mean square) function, [3-187](#)
 - rtti (enable run-time type identification) C++ mode
 - compiler switch, [1-56](#)
 - run-time
 - disabling type identification, [1-56](#)
 - enabling type identification, [1-56](#)
- S
- S (stop after compilation) compiler switch, [1-44](#)
 - s (strip debugging information)
 - compiler switch, [1-45](#)
 - save-temps (save intermediate files)
 - compiler switch, [1-45](#)
 - scratch registers, [1-204](#), [1-214](#)
 - search path
 - for include files, [1-32](#)
 - for library files, [1-33](#)
 - searching for #included files, [1-184](#)
 - section elimination, [2-37](#)
 - SEG_ARGV memory section, [1-197](#)
 - segment (*see* placement support keyword (segment))

INDEX

- separate fraction and exponent
function, [3-128](#)
- set header file, [3-35](#)
- setjmp.h, [3-13](#)
- setvbuf function, [3-16](#)
- show (display command line)
compiler switch, [1-45](#)
- sideways sum functions
(int4x16 values), [1-110](#)
int2x16 values, [1-106](#)
- sig argument, [3-148](#), [3-190](#)
- sign (sign) function, [3-189](#)
- signal (define signal handling)
function, [3-190](#)
- signal handlers
non-reentrant, [3-192](#)
- signal handling, [3-13](#)
- signal.h header file, [3-13](#)
- siglfnr function, [3-190](#), [3-191](#)
- siglnr function, [3-192](#)
- signals function, [3-190](#), [3-191](#)
- signed-bitfield (make plain bitfields
signed) compiler switch, [1-46](#)
- signed-char (make char signed)
compiler switch, [1-46](#)
- silicon revision
setting, [1-45](#)
- simulator
library support, [3-5](#)
services, [3-25](#)
- sin (sine) function, [3-193](#)
- single case range, [1-176](#)
- sinh (hyperbolic sine) function,
[3-194](#)
- si-revision (silicon revision)
compiler switch, [1-45](#)
- sizeof operator, [1-70](#), [1-176](#)
- source code
library, [3-7](#)
- sprintf function, [3-17](#)
- sqrt (square root) function, [3-195](#)
- srand (random number seed)
function, [3-196](#)
- sstream header file, [3-32](#)
- stack
header file, [3-35](#)
pointer, [1-187](#)
- stack frame, [1-185](#)
free space, [1-189](#)
incoming arguments, [1-188](#)
linkage information, [1-188](#)
local variables/temporaries, [1-188](#)
outgoing arguments, [1-189](#)
outgoing linkage, [1-189](#)
- stage count, [2-66](#)
- start-up files
C, [3-5](#)
C++, [3-5](#)
- STAT registers, [1-129](#)
- statement expression, [1-171](#)
- Static Condition Flags register,
[1-209](#)
- statistical
functions, [3-27](#)
- stats.h header file, [3-27](#)
- stdarg.h header file, [3-14](#)

- stddef.h header file, [3-14](#)
- stdexcept header file, [3-32](#)
- stdio functions, [1-197](#)
- stdio.h header file, [1-197](#), [3-14](#)
- stdlib.h header file, [3-21](#)
- stdout output stream, [1-28](#)
- streambuf header file, [3-32](#)
- stride argument
 - FFT function, [3-24](#)
- string
 - header file, [3-32](#)
 - literals with line breaks, [1-176](#)
- string to double conversion, [3-197](#)
- string to float representation, [3-199](#)
- string to integer conversion, [3-201](#)
- string to long double conversion, [3-203](#)
- string to long integer conversion, [3-202](#)
- string to long long integer
 - conversion, [3-205](#)
- string to unsigned long integer
 - conversion, [3-206](#)
- string to unsigned long long integer
 - conversion, [3-207](#)
- string.h header file, [3-22](#)
- strstream header file, [3-33](#)
- strtod (convert string to double)
 - function, [3-197](#)
- strtof (string to float) function, [3-199](#)
- strtoi (string to integer conversion)
 - function, [3-201](#)
- strtol (string to long integer)
 - function, [3-202](#)
- strtold (string to long double
 - conversion) function, [3-203](#)
- strtold function, [3-203](#)
- strtoll (string to long long integer)
 - function, [3-205](#)
- strtoul (string to unsigned long
 - integer conversion) function, [3-206](#)
- strtoull (string to unsigned long
 - long integer) function, [3-207](#)
- sub-word data types, [2-9](#)
- switches
 - C/C++ mode selection, [1-21](#)
 - compiler common, [1-22](#) to [1-53](#)
 - @ filename (command file), [1-22](#)
 - A (assert) compiler switch, [1-22](#)
 - align-branch-lines, [1-23](#)
 - bss (placing data in bss), [1-24](#)
 - build-lib (build library), [1-24](#)
 - C (comments), [1-24](#)
 - c (compile only), [1-24](#)
 - char-size-8|32, [1-25](#)
 - char-size-any, [1-24](#)
 - const-read-write, [1-25](#)
 - debug-types, [1-26](#)
 - default-branch-(np|p), [1-26](#)
 - default-linkage-asm (assembler linkage), [1-26](#)
 - default-linkage-C (C linkage), [1-26](#)
 - default-linkage-C++ (C++ link-

- age), [1-26](#)
- Dmacro (define macro), [1-25](#)
- double-size-32|64, [1-27](#)
- double-size-any, [1-27](#)
- dry (verbose dry-run), [1-28](#)
- dryrun (terse dry-run), [1-28](#)
- E (stop after preprocessing), [1-28](#)
- ED (run after preprocessing to file), [1-29](#)
- EE (run after preprocessing), [1-29](#)
- extra-keywords (enable short-form keywords), [1-29](#)
- flags (command-line input), [1-29](#)
- force-circbuf, [1-30](#)
- fp-associative (floating-point associative operation), [1-30](#)
- full-version (display versions), [1-30](#)
- g (generate debug information), [1-30](#)
- H (list headers), [1-31](#)
- h[elp] (command-line help), [1-31](#)
- HH (list headers and compile), [1-31](#)
- I- (start include directory), [1-31](#)
- I directory (include search directory), [1-32](#)
- include (include file), [1-32](#)
- ipa (interprocedural analysis), [1-33](#)
- l (link library), [1-33](#)
- L directory (library search directory), [1-33](#)
- M (generate make rules only), [1-33](#)
- map filename (generate a memory map), [1-35](#)
- MD (generate make rules and compile), [1-34](#)
- mem (enable memory initialization), [1-35](#)
- MM (generate make rules and compile), [1-34](#)
- Mo (processor output file), [1-34](#)
- MQ, [1-34](#)
- Mt (output make rules), [1-34](#)
- no-align-branch-lines, [1-35](#)
- no-alttok, [1-35](#)
- no-annotate (disable alternative tokens), [1-35](#)
- no-bss, [1-36](#)
- no-builtin (no built-in functions), [1-36](#)
- no-circbuf (no circular buffer), [1-36](#)
- no-const-strings, [1-36](#)
- no-defs (disable defaults), [1-37](#)
- no-extra-keywords (disable short-form keywords), [1-37](#)
- no-fp-associative, [1-37](#)
- no-mem (disable memory initialization), [1-37](#)

- no-saturation (no faster operations), [1-37](#)
- no-std-ass (disable standard assertions), [1-38](#)
- no-std-def (disable standard macro definitions), [1-38](#)
- no-std-inc (disable standard include search), [1-38](#)
- no-std-lib (disable standard library search), [1-38](#)
- nothreads (disable thread-safe build), [1-38](#)
- O (enable optimizations), [1-39](#)
- o filename (output file), [1-40](#)
- Oa (automatic function inlining), [1-39](#)
- Os (optimize for size), [1-39](#)
- Ov (optimize for speed vs. size), [1-40](#)
- P (omit line numbers), [1-40](#)
- path (tool location), [1-40](#)
- path-install directory (installation location), [1-41](#)
- path-output directory (non-temporary files location), [1-41](#)
- path-temp directory (temporary files location), [1-41](#)
- pch (precompiled header), [1-41](#)
- pchdir (locate PCHRepository), [1-42](#)
- pedantic (ANSI standard warnings), [1-42](#)
- pedantic-errors (ANSI C errors), [1-42](#)
- pguide (profile-guided optimization), [1-42](#)
- PP (omit line numbers and compile), [1-40](#)
- pplist file (preprocessor listing), [1-42](#)
- proc identifier (processor), [1-43](#)
- R- (disable source path), [1-44](#)
- R directory (add source directory), [1-44](#)
- S (stop after compilation), [1-44](#)
- s (strip debugging information), [1-45](#)
- save-temps (save intermediate files), [1-45](#)
- show (display command line), [1-45](#)
- signed-bitfield (make plain bitfields signed), [1-46](#)
- signed-char (make char signed), [1-46](#)
- si-revision version (silicon revision), [1-45](#)
- sourcefile (sourcefile parameter), [1-22](#)
- syntax-only (just check syntax), [1-46](#)
- syntax-only (system definitions), [1-47](#)
- T (linker description file), [1-47](#)
- threads (enable thread-safe

- build), 1-48
- time (time the compiler), 1-48
- Umacro (undefine macro), 1-48
- unsigned-bitfield, 1-48
- unsigned-char (make char unsigned), 1-49
- v (version and verbose), 1-49
- val-global (add global names), 1-49
- vcse-add, 1-50
- vcse-cname, 1-50
- vcse-enforce, 1-50
- vcse-names, 1-50
- verbose (display command line), 1-50
- version (display version), 1-51
- w (disable all warnings), 1-51
- warn-protos (warn if incomplete prototype), 1-52
- Wdriver-limit (maximum process errors), 1-51
- Werror-limit (maximum switches), 1-51
- workaround, 1-52
- Wremarks (enable diagnostic warnings), 1-51
- write-files (enable file redirection), 1-52
- write-opts (user options), 1-53
- Wterse (enable terse warnings), 1-52
- xref file (cross-reference list), 1-53

- syntax-only (just check syntax)
 - compiler switch, 1-46
- sysdef (system definitions)
 - compiler switch, 1-47
- sysreg.h header file, 2-34
- system header files, 1-181

T

- T (linker description file) compiler switch, 1-47
- tan (tangent) function, 3-209
- tanh (hyperbolic tangent) function, 3-210
- template
 - asm() in C programs, 1-76
 - assembly construct, 1-76
 - instantiation pragmas, 1-165
- threads (enable thread-safe build)
 - compiler switch, 1-48
- time (time the compiler) switch, 1-48
- time.h header file, 3-22
- transpm (real matrix transpose)
 - function, 3-211
- trip
 - count, 2-67
 - loop count, 2-60
 - maximum, 2-67
 - minimum, 2-67
 - modulo, 2-67
- true (*see* Boolean type support
 - keywords (bool, true, false))
- ts_exit_.dlb C exit routine files, 3-5

- ts_exit_cpp_.dlb C++ exit routine files, [3-6](#)
- ts_hdr_.dlb C start-up files, [3-5](#)
- ts_hdr_cpp_.dlb start-up files for C++, [3-5](#)
- twiddle table, [3-24](#)
- twidfft function, [3-212](#)
- twidfft function, [3-214](#)
- two-dimensional FFT, [3-79](#), [3-181](#)
- type
 - cast, [1-176](#)
- typeof keyword, [1-172](#)

U

- U (undefine macro) compiler switch, [1-25](#), [1-48](#)
- unclobbered registers, [1-160](#)
- unsigned-bitfield (make plain bitfields unsigned) compiler switch, [1-48](#)
- unsigned-char (make char unsigned) compiler switch, [1-49](#)
- user header files, [1-181](#)
- utility header file, [3-36](#)

V

- v (version and verbose) compiler switch, [1-49](#)
- val-global (add global names) compiler switch, [1-49](#)
- var (variance) function, [3-216](#)
- varargs routines, [1-192](#)
- variable

- argument macros, [1-175](#)
- length array, [1-175](#)
- length array support, [1-94](#)
- VCSE components
 - defining, [1-50](#)
- vcse_enforce utility
 - invoking, [1-50](#)
- vcse-add compiler switch, [1-50](#)
- vcse-cname (component name) compiler switch, [1-50](#)
- vcse-enforce compiler switch, [1-50](#)
- vcse-names compiler switch, [1-50](#)
- vecdot (real vector dot product) function, [3-217](#)
- vecsadd (real vector + scalar addition) function, [3-218](#)
- vecsmult (real vector * scalar multiplication) function, [3-219](#)
- vecssub (real vector - scalar subtraction) function, [3-220](#)
- vector functions, [3-28](#)
- vector header file, [3-36](#)
- vector.h header file, [3-28](#)
- vectorization
 - annotations, [2-64](#)
 - avoiding, [2-43](#)
 - factor, [2-60](#)
 - loop, [2-43](#)
 - unroll and jam, [2-61](#)
- vecvadd (real vector + vector addition) function, [3-221](#)
- vecvmult (real vector * vector multiplication) function, [3-222](#)

INDEX

vecvsub (real vector - vector subtraction) function, [3-223](#)
-verbose (display command line) compiler switch, [1-50](#)
-version (display version) switch, [1-51](#)
VIDL source text, [1-184](#)
VisualDSP++
 IDDE, [1-3](#), [1-9](#)
 Kernel (VDK), [1-48](#)
 running compiler, [1-2](#)
 setting compiler options, [1-9](#)
 simulator, [1-198](#)
VisualDSP++ compiler C/C++
 language extensions, [1-66](#), [1-67](#)
volatile and asm() C program constructs, [1-85](#)

W

-w (disable all warnings) switch, [1-51](#)
-Warn-protos (warn if incomplete prototype) compiler switch, [1-52](#)
-Wdriver-limit (maximum process errors) compiler switch, [1-51](#)
-Werror-limit (maximum compiler errors) compiler switch, [1-51](#)

window generators, [3-29](#)
window.h header file, [3-29](#)
word addressing mode
 selecting, [1-3](#), [1-25](#)
-workaround compiler switch, [1-52](#)
-Wremarks (enable diagnostic warnings) compiler switch, [1-51](#)
-write-files (enable driver I/O redirection) compiler switch, [1-52](#)
-write-opts (enable driver I/O pipe) compiler switch, [1-53](#)
writes
 array element, [2-26](#)
writing preprocessor macros, [1-182](#)
-Wterse (enable terse warnings) compiler switch, [1-52](#)

X

-xref (cross-reference list) compiler switch, [1-53](#)

Z

zero length arrays, [1-175](#)
zero_cross (count zero crossings) function, [3-224](#)