VisualDSP++® 4.0
C/C++ Compiler and Library Manual for SHARC® Processors

Revision 5.0, January 2005

Part Number
82-001963-01

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

ANALOG DEVICES

Copyright Information

©2005 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, 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 of This Manual xxxi
Intended Audience xxxi
Manual Contents xxxii
What's New in This Manual xxxiii
Technical or Customer Support xxxiii
Supported Processors xxxiv
Product Information xxxiv
MyAnalog.com xxxiv
Processor Product Information xxxv
Related Documents xxxvi
Online Technical Documentation xxxvii
Accessing Documentation From VisualDSP++ xxxvii
Accessing Documentation From Windows xxxviii
Accessing Documentation From the Web xxxviii
Printed Manuals xxxix
VisualDSP++ Documentation Set xxxix
Hardware Tools Manuals xxxix
Processor Manuals xxxix

Data Sheets xxxix
Notation Conventions xl

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 1-22
-c89 1-22
-c++ 1-22
C/C++ Compiler Common Switch Descriptions 1-23
sourcefile 1-23
-@ filename 1-23
-A name[tokens] 1-23
-aligned-stack 1-24
-alttok 1-24
-annotate-loop-instr 1-25
-bss 1-25
-build-lib 1-25
-C 1-25
-c 1-26
-compatible-pm-dm 1-26

-const-read-write 1-26
-Dmacro[=definition] 1-26
-debug-types 1-26
-default-linkage[-asm | -C | -C++] 1-27
-double-size[-32|-64] 1-27
-double-size-any 1-28
-dry 1-28
-dryrun 1-28
-E 1-29
-ED 1-29
-EE 1-29
-extra-keywords 1-29
-flags-{asm|compiler|lib|link|mem} switch [,switch2 [,...]] 1-29
-float-to-int 1-30
-force-circbuf 1-30
-fp-associative 1-30
-full-version 1-30
-g 1-31
-H 1-31
-HH 1-31
-h[elp] 1-32
-I directory [{,|;} directory...] 1-32
-I- 1-32
-i 1-33

-include filename 1-33
-ipa 1-33
-L directory[{;|,}directory…] 1-34
-l library 1-34
-M 1-35
-MD 1-35
-MM 1-35
-Mo _filename_ 1-35
-Mt filename 1-35
-MQ 1-36
-map filename 1-36
-mem 1-36
-multiline 1-36
-no-aligned-stack 1-36
-no-alttok 1-36
-no-annotate 1-37
-no-annotate-loop-instr 1-37
-no-bss 1-37
-no-builtin 1-37
-no-circbuf 1-38
-no-db 1-38
-no-defs 1-38
-no-extra-keywords 1-38
-no-fp-associative 1-38

-no-mem 1-39
-no-multiline 1-39
-no-saturation 1-39
-no-simd 1-39
-no-std-ass 1-39
-no-std-def 1-40
-no-std-inc 1-40
-no-std-lib 1-40
-no-threads 1-40
-O[0|1] 1-40
-Oa 1-41
-Og 1-41
-Os 1-41
-Ov num 1-41
-o filename 1-42
-P 1-42
-PP 1-42
-path-{ asm | compiler | lib | link | mem } directory 1-42
-path-install directory 1-42
-path-output directory 1-43
-path-temp directory 1-43
-pch 1-43
-pchdir directory 1-43
-pedantic 1-43

-pedantic-errors 1-44
-pguide 1-44
-pplist filename 1-44
-proc processor 1-45
-progress-rep-func 1-45
-progress-rep-gen-opt 1-45
-progress-rep-mc-opt 1-46
-R directory[{:|,}directory …] 1-46
-R- 1-47
-reserve register[, register …] 1-47
-restrict-hardware-loops <maximum> 1-47
-S 1-47
-s 1-48
-save-temps 1-48
-section <id>=<section_name>[,<id>=section_name>] ... 1-48
-show 1-48
-si-revision version 1-49
-signed-bitfield 1-50
-signed-char 1-50
-structs-do-not-overlap 1-51
-switch-pm 1-51
-syntax-only 1-51
-sysdefs 1-52
-T filename 1-52

-threads 1-52
-time 1-53
-Umacro 1-53
-unsigned-bitfield 1-53
-unsigned-char 1-54
-v 1-54
-verbose 1-54
-version 1-54
-W[error|remark|suppress|warn] number[,number ...] 1-55
-Wdriver-limit number 1-55
-Werror-limit number 1-55
-Wremarks 1-55
-Wterse 1-55
-w 1-56
-warn-protos 1-56
-workaround <workaround>[,<workaround>]* 1-56
-write-files 1-56
-write-opts 1-57
-xref <filename> 1-57
C++ Mode Compiler Switch Descriptions 1-58
-anach 1-58
-eh 1-59
-no-anach 1-60
-no-demangle 1-60

-no-eh 1-60
-no-implicit-inclusion 1-60
-no-rtti 1-60
-rtti 1-61
Data Type and Data Type Sizes 1-61
Integer Data Types 1-62
Floating-Point Data Types 1-62
Environment Variables Used by the Compiler 1-63
Optimization Control 1-64
Interprocedural Analysis 1-67
Interaction with Libraries 1-68
C/C++ Compiler Language Extensions 1-70
Inline Function Support Keyword (inline) 1-74
Inline Assembly Language Support Keyword (asm) 1-75
Assembly Construct Template 1-77
asm() Construct Syntax Rules 1-78
asm() Construct Template Example 1-79
Assembly Construct Operand Description 1-80
Assembly Constructs With Multiple Instructions 1-86
Assembly Construct Reordering and Optimization 1-87
Assembly Constructs With Input and Output Operands 1-87
Assembly Constructs and Flow Control 1-88
Guildelines on the Use of asm() Statements 1-89
Dual Memory Support Keywords (pm dm) 1-90

Memory Keywords and Assignments/Type Conversions 1-92
Memory Keywords and Function Declarations/Pointers 1-93
Memory Keywords and Function Arguments 1-93
Memory Keywords and Macros 1-94
Placement Support Keyword (section) 1-95
Placement of Compiler-Generated Data 1-95
Boolean Type Support Keywords (bool, true, false) 1-96
Pointer Class Support Keyword (restrict) 1-96
Variable-Length Array Support 1-97
Non-Constant Initializer Support 1-99
Indexed Initializer Support 1-99
Aggregate Constructor Expression Support 1-101
Preprocessor Generated Warnings 1-102
C++ Style Comments 1-102
Compiler Built-In Functions 1-102
Access to System Registers 1-103
Circular Buffer Built-In Functions 1-107
Circular Buffer Increment of an Index 1-107
Circular Buffer Increment of a Pointer 1-107
Compiler Performance Built-in Functions 1-108
Pragmas 1-110
Data Alignment Pragmas 1-111
#pragma align num 1-112
#pragma alignment_region (alignopt) 1-113

#pragma pack (alignopt) 1-114
#pragma pad (alignopt) 1-114
Interrupt Handler Pragma 1-115
Loop Optimization Pragmas 1-116
#pragma SIMD_for 1-116
#pragma all_aligned 1-117
#pragma no_vectorization 1-117
#pragma loop_count(min, max, modulo) 1-117
#pragma no_alias 1-118
#pragma vector_for 1-118
General Optimization Pragmas 1-119
Function Side-Effect Pragmas 1-120
#pragma alloc 1-121
#pragma const 1-121
#pragma noreturn 1-122
#pragma pure 1-122
#pragma regs_clobbered string 1-123
#pragma result_alignment (n) 1-127
Template Instantiation Pragmas 1-127
#pragma instantiate instance 1-128
#pragma do_not_instantiate instance 1-128
#pragma can_instantiate instance 1-129
Header File Control Pragmas 1-129
#pragma hdrstop 1-129

#pragma no_implicit_inclusion 1-130
#pragma no_pch 1-131
#pragma once 1-131
#pragma system_header 1-132
Linking Control Pragmas 1-132
#pragma linkage_name identifier 1-132
#pragma retain_name 1-132
#pragma section/#pragma default_section 1-133
#pragma weak_entry 1-135
Diagnostic Control Pragmas 1-135
Modifying the Severity of Specific Diagnostics 1-135
Modifying the Behavior of an Entire Class of Diagnostics 1-136
Saving or Restoring the Current Behavior of All Diagnostics .. 1-137
Code Generation Pragmas 1-138
GCC Compatibility Extensions 1-138
Statement Expressions 1-138
Type Reference Support Keyword (Typeof) 1-140
GCC Generalized Lvalues 1-141
Conditional Expressions with Missing Operands 1-141
Hexadecimal Floating-Point Numbers 1-142
Zero-Length Arrays 1-142
Variable Argument Macros 1-143
Line Breaks in String Literals 1-143
Arithmetic on Pointers to Void and Pointers to Functions 1-143

Cast to Union 1-144
Ranges in Case Labels 1-144
Declarations Mixed With Code 1-144
Escape Character Constant 1-145
Alignment Inquiry Keyword (__alignof__) 1-145
Keyword for Specifying Names in Generated Assembler (asm) 1-145
Function, Variable and Type Attribute Keyword (__attribute__) 1-146
C++ Fractional Type Support 1-146
Format of Fractional Literals 1-147
Conversions Involving Fractional Values 1-147
Fractional Arithmetic Operations 1-148
Mixed Mode Operations 1-148
Saturated Arithmetic 1-149
SIMD Support 1-150
Using SIMD Mode with Multichannel Data 1-151
Using SIMD Mode with Single-Channel Data 1-152
Restrictions to Using SIMD 1-154
SIMD_for Pragma Syntax 1-155
Compiler Constraints on Using SIMD C/C++ 1-157
Impact of Anomaly #40 on SIMD 1-158
Performance When Using SIMD C/C++ 1-159
Examples Using SIMD C 1-160
Using SIMD C (Problem Cases—Data Increments) 1-160
Using SIMD C (Problem Cases—Data Alignment) 1-162

Preprocessor Features 1-165
Predefined Preprocessor Macros 1-166
__2106x__ 1-166
__2116x__ 1-166
__2126x__ 1-166
__2136x__ 1-166
__ADSP21000__ 1-166
__ADSP21020__ 1-167
__ADSP21060__ 1-167
__ADSP21061__ 1-167
__ADSP21062__ 1-167
__ADSP21065L__ 1-167
__ADSP21160__ 1-167
__ADSP21161__ 1-167
__ADSP21261__ 1-168
__ADSP21262__ 1-168
__ADSP21266__ 1-168
__ADSP21267__ 1-168
__ADSP21363__ 1-168
__ADSP21364__ 1-168
__ADSP21365__ 1-168
__ADSP21366__ 1-169
__ADSP21367__ 1-169
__ADSP21368__ 1-169

__ADSP21369__ 1-169
__ANALOG_EXTENSIONS__ 1-169
__cplusplus 1-169
__DATE__ 1-169
__DOUBLES_ARE_FLOATS__ 1-170
__ECC__ 1-170
__EDG__ 1-170
__EDG_VERSION__ 1-170
__FILE__ 1-170
__LINE__ 1-170
_NO_LONGLONG 1-170
__NO_BUILTIN 1-171
__SIGNED_CHARS__ 1-171
__SIMDSHARC__ 1-171
__STDC__ 1-171
__STDC_VERSION__ 1-171
__TIME__ 1-171
__VERSION__ 1-171
__VERSIONNUM__ 1-172
__WORKAROUNDS_ENABLED 1-172
Writing Macros 1-172
C/C++ Run-Time Model and Environment 1-175
C/C++ Run-Time Environment 1-175
Memory Usage 1-177

Measuring the Performance of C Compiler 1-183
Support for argv/argc 1-185
Using Multiple Heaps 1-185
Declaring a Heap 1-186
Heap Identifiers 1-189
Using Alternate Heaps with the Standard Interface 1-189
Using the Alternate Heap Interface 1-190
Example C Programs 1-191
Compiler Registers 1-192
Miscellaneous Information About Registers 1-193
User Registers 1-193
Call Preserved Registers 1-194
Scratch Registers 1-195
Stack Registers 1-196
Alternate Registers 1-196
Managing the Stack 1-197
Transferring Function Arguments and Return Value 1-202
Using Data Storage Formats 1-205
Using the Run-Time Header 1-208
C/C++ and Assembly Interface 1-209
Calling Assembly Subroutines from C/C++ Programs 1-209
Calling C/C++ Functions from Assembly Programs 1-211
Using Mixed C/C++ and Assembly Support Macros 1-214
entry 1-214

exit 1-214
leaf_entry 1-215
leaf_exit 1-215
ccall(x) 1-215
reads(x) 1-215
puts=x 1-215
gets(x) 1-215
alter(x) 1-216
save_reg 1-216
restore_reg 1-216
Using Mixed C/C++ and Assembly Naming Conventions 1-218
Implementing C++ Member Functions in Assembly Language 1-220
Writing C/C++ Callable SIMD Subroutines 1-222
C++ Programming Examples 1-223
Using Fract Support 1-223
Using Complex Support 1-224
Mixed C/C++/Assembly Programming Examples 1-226
Using Inline Assembly (Add) 1-227
Using Macros to Manage the Stack 1-228
Using Scratch Registers (Dot Product) 1-229
Using Void Functions (Delay) 1-231
Using the Stack for Arguments and Return (Add 5) 1-232
Using Registers for Arguments and Return (Add 2) 1-233
Using Non-Leaf Routines That Make Calls (RMS) 1-234

Using Call Preserved Registers (Pass Array) 1-236
Compiler C++ Template Support 1-238
Template Instantiation 1-238
Identifying Uninstantiated Templates 1-239

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 the Statistical Profiler 2-5
Using Profile-Guided Optimization 2-6
Using Interprocedural Optimization 2-7
Data Types 2-8
Avoiding Emulated Arithmetic 2-9
Getting the Most From IPA 2-10
Initialize Constants Statically 2-10
Dual Word-Aligning Your Data 2-11
Using __builtin_aligned 2-12
Avoiding Aliases 2-13
Indexed Arrays Versus Pointers 2-15
Trying Pointer and Indexed Styles 2-16
Function Inlining 2-16
Using Inline asm Statements 2-18
Memory Usage 2-19

Improving Conditional Code 2-21
Loop Guidelines 2-22
Keeping Loops Short 2-22
Avoiding Unrolling Loops 2-22
Avoiding Loop-Carried Dependencies 2-23
Avoiding Loop Rotation by Hand 2-24
Avoiding Array Writes in Loops 2-25
Inner Loops vs. Outer Loops 2-26
Avoiding Conditional Code in Loops 2-26
Avoiding Placing Function Calls in Loops 2-27
Avoiding Non-Unit Strides 2-27
Loop Control 2-28
Using the Restrict Qualifier 2-29
Using the Const Qualifier 2-30
Avoiding Long Latencies 2-31
Using Built-In Functions in Code Optimization 2-32
System Support Built-In Functions 2-32
Using Circular Buffers 2-33
Smaller Applications: Optimizing for Code Size 2-36
Using Pragmas for Optimization 2-38
Function Pragmas 2-38
#pragma alloc 2-38
#pragma const 2-39
#pragma pure 2-39

#pragma result_alignment 2-40
#pragma regs_clobbered 2-40
#pragma optimize_{off|for_speed|for_space|as_cmd_line} .. 2-42
Loop Optimization Pragmas 2-43
#pragma loop_count 2-43
#pragma no_vectorization 2-43
#pragma vector_for 2-44
#pragma SIMD_for 2-45
#pragma all_aligned 2-45
#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
Loop Flattening 2-61
Vectorization Annotations 2-62
Modulo Scheduling 2-65

C/C++ RUN-TIME LIBRARY

C and C++ Run-Time Libraries Guide 3-3
Calling Library Functions 3-3

Linking Library Functions 3-4
Working with Library Header Files 3-12
assert.h 3-13
ctype.h 3-13
cycle_count.h 3-13
cycles.h 3-13
device.h 3-14
device_int.h 3-14
errno.h 3-14
float.h 3-15
iso646.h 3-15
limits.h 3-16
locale.h 3-16
math.h 3-16
setjmp.h 3-18
signal.h 3-18
Interrupt Dispatchers 3-18
Interrupts and Circular Buffering 3-21
Avoiding Self-Modifying Code 3-21
Interrupt Nesting Restrictions on ADSP-2116x/2126x/2136x Processors 3-22
Restriction on Use of Super-Fast Dispatcher on ADSP-2106x Processors 3-22
stdarg.h 3-23
stddef.h 3-23

stdio.h 3-24
stdlib.h 3-26
string.h 3-27
time.h 3-27
Calling Library Functions from an ISR 3-29
Using Compiler Built-In C Library Functions 3-30
Abridged C++ Library Support 3-32
Embedded C++ Library Header Files 3-32
complex 3-32
exception 3-33
fract 3-33
fstream 3-33
iomanip 3-33
ios 3-33
iosfwd 3-34
iostream 3-34
istream 3-34
new 3-34
ostream 3-34
sstream 3-34
stdexcept 3-34
streambuf 3-35
string 3-35
strstream 3-35

C++ Header Files for C Library Facilities 3-35
Embedded Standard Template Library Header Files 3-37
algorithm 3-37
deque 3-37
functional 3-37
hash_map 3-37
hash_set 3-37
iterator 3-37
list 3-37
map 3-38
memory 3-38
numeric 3-38
queue 3-38
set 3-38
stack 3-38
utility 3-38
vector 3-38
fstream.h 3-39
iomanip.h 3-39
iostream.h 3-39
new.h 3-39
Using the Thread-Safe C/C++ Run-Time Libraries with VDK 3-39
Measuring Cycle Counts 3-40
Basic Cycle Counting Facility 3-40

Cycle Counting Facility with Statistics 3-42
Using time.h to Measure Cycle Counts 3-45
Determining the Processor Clock Rate 3-47
Considerations When Measuring Cycle Counts 3-47
File I/O Support 3-49
Extending I/O Support To New Devices 3-50
DevEntry Structure 3-50
Registering New Devices 3-55
Pre-Registering Devices 3-56
Default Device 3-57
Remove and Rename Functions 3-58
Default Device Driver Interface 3-58
Data Packing For Primitive I/O 3-59
Data Structure for Primitive I/O 3-60
C Run-Time Library Reference 3-64
abort 3-65
abs 3-66
acos 3-67
asctime 3-68
asin 3-70
atan 3-71
atan2 3-72
atexit 3-73
atof 3-74

atoi 3-77
atol 3-78
atold 3-79
avg 3-82
bsearch 3-83
calloc 3-85
ceil 3-86
circindex 3-87
circptr 3-89
clear_interrupt 3-92
clip 3-101
clock 3-102
cos 3-103
cosh 3-104
count_ones 3-105
ctime 3-106
difftime 3-107
div 3-108
exit 3-109
exp 3-110
fabs 3-111
floor 3-112
fmod 3-113
free 3-114

frexp 3-115
getenv 3-116
gmtime 3-117
heap_calloc 3-118
heap_free 3-120
heap_install 3-122
heap_lookup_name 3-125
heap_malloc 3-127
heap_realloc 3-129
interrupt 3-132
isalnum 3-134
isalpha 3-135
iscntrl 3-136
isdigit 3-137
isgraph 3-138
isinf 3-139
islower 3-141
isnan 3-142
isprint 3-144
ispunct 3-145
isspace 3-146
isupper 3-147
isxdigit 3-148
labs 3-149

lavg 3-150
lclip 3-151
lcount_ones 3-152
ldexp 3-153
ldiv 3-154
lmax 3-155
lmin 3-156
localeconv 3-157
localtime 3-160
log 3-162
log10 3-163
longjmp 3-164
malloc 3-166
max 3-167
memchr 3-168
memcmp 3-169
memcpy 3-170
memmove 3-171
memset 3-172
min 3-173
mktime 3-174
modf 3-176
pow 3-177
qsort 3-178

raise 3-180
rand 3-181
realloc 3-182
setjmp 3-184
setlocale 3-185
set_alloc_type 3-186
signal 3-188
sin 3-190
sinh 3-191
sqrt 3-192
srand 3-193
strcat 3-194
strchr 3-195
strcmp 3-196
strcoll 3-197
strcpy 3-198
strcspn 3-199
strerror 3-200
strftime 3-201
strlen 3-205
strncat 3-206
strncmp 3-207
strncpy 3-208
strpbrk 3-209

strrchr 3-210
strspn 3-211
strstr 3-212
strtod 3-213
strtold 3-216
strtok 3-219
strtol 3-221
strtoul 3-223
strxfrm 3-225
system 3-227
tan 3-228
tanh 3-229
time 3-230
tolower 3-231
toupper 3-232
va_arg 3-233
va_end 3-235
va_start 3-236

DSP LIBRARY FOR ADSP-2106X AND ADSP-21020 PROCESSORS

DSP Run-Time Library Guide 4-2
Calling DSP Library Functions 4-2
Linking DSP Library Functions 4-3
Working With Library Source Code 4-3

DSP Header Files 4-4
21020.h — ADSP-21020 DSP Functions 4-4
21060.h — ADSP-2106x DSP Functions 4-4
21065L.h — ADSP-21065L DSP Functions 4-5
asm_sprt.h — Mixed C/Assembly Support 4-5
cmatrix.h — Complex Matrix Functions 4-5
comm.h — A-law and μ-law Companders 4-5
complex.h — Basic Complex Arithmetic Functions 4-5
cvector.h — Complex Vector Functions 4-6
Header Files Defining Processor-Specific System Register Bits 4-6
Header Files To Allow Access to Memory Mapped Registers From C/C++ Code 4-7
dma.h — DMA Support Functions 4-7
filters.h — DSP Filters 4-8
macros.h — Circular Buffers 4-8
math.h — Math Functions 4-8
matrix.h — Matrix Functions 4-9
saturate.h — Saturation Mode Arithmetic 4-10
sport.h — Serial Port Support Functions 4-10
stats.h — Statistical Functions 4-10
sysreg.h — Register Access 4-10
trans.h — Fast Fourier Transforms 4-10
vector.h — Vector Functions 4-11
window.h — Window Generators 4-11
Built-In DSP Functions 4-13

DSP Run-Time Library Reference 4-15
a_compress 4-16
a_expand 4-17
alog 4-18
alog10 4-19
arg 4-20
autocoh 4-21
autocorr 4-23
biquad 4-25
cabs 4-28
cadd 4-29
cartesian 4-30
cdiv 4-32
cexp 4-33
cfftN 4-34
cmatmadd 4-37
cmatmmlt 4-39
cmatmsub 4-41
cmatsadd 4-43
cmatsmlt 4-45
cmatssub 4-47
cmlt 4-49
conj 4-50
convolve 4-51

copysign 4-53
cot 4-54
crosscoh 4-55
crosscorr 4-57
csub 4-59
cvecdot 4-60
cvecsadd 4-62
cvecsmlt 4-64
cvecssub 4-66
cvecvadd 4-68
cvecvmlt 4-70
cvecvsub 4-72
favg 4-74
fclip 4-75
fir 4-76
fmax 4-78
fmin 4-79
gen_bartlett 4-80
gen_blackman 4-82
gen_gaussian 4-83
gen_hamming 4-84
gen_hanning 4-85
gen_harris 4-86
gen_kaiser 4-87

gen_rectangular 4-89
gen_triangle 4-90
gen_vonhann 4-92
histogram 4-93
idle 4-95
ifftN 4-96
iir 4-99
matinv 4-102
matmadd 4-103
matmmlt 4-105
matmsub 4-107
matsadd 4-109
matsmlt 4-111
matssub 4-113
mean 4-115
mu_compress 4-116
mu_expand 4-117
norm 4-118
polar 4-119
poll_flag_in 4-121
rfftN 4-123
rms 4-126
rsqrt 4-127
set_flag 4-128

set_semaphore 4-130
timer_off 4-131
timer0_off, timer1_off 4-132
timer_on 4-133
timer0_on, timer1_on 4-134
timer_set 4-135
timer0_set, timer1_set 4-137
transpm 4-139
var 4-141
vecdot 4-142
vecsadd 4-144
vecsmlt 4-145
vecssub 4-146
vecvadd 4-147
vecvmlt 4-149
vecvsub 4-150
zero_cross 4-152

DSP LIBRARY FOR ADSP-2116X/2126X/2136X PROCESSORS

DSP Run-Time Library Guide 5-2
Calling DSP Library Functions 5-2
Linking DSP Library Functions 5-3
Working With Library Source Code 5-4
DSP Header Files 5-5

21160.h – ADSP-2116x DSP Functions 5-5
21262.h – ADSP-21262 DSP Functions 5-5
21363.h – ADSP-21363 DSP Functions 5-5
21364.h – ADSP-21364 DSP Functions 5-5
21365.h – ADSP-21365 DSP Functions 5-6
asm_sprt.h – Mixed C/Assembly Support 5-6
cmatrix.h – Complex Matrix Functions 5-6
comm.h – A-law and μ-law Companders 5-6
complex.h – Basic Complex Arithmetic Functions 5-7
cvector.h – Complex Vector Functions 5-7
Header Files Defining Processor-Specific System Register Bits 5-8
Header Files To Allow Access to Memory Mapped Registers From C/C++ Code 5-8
dma.h – DMA Support Functions 5-9
filter.h – DSP Filters and Transformations 5-9
filters.h – DSP Filters 5-11
macro.h – Circular Buffers 5-11
math.h – Math Functions 5-11
matrix.h – Matrix Functions 5-13
saturate.h – Saturation Mode Arithmetic 5-13
sport.h – Serial Port Support Functions 5-13
stats.h – Statistical Functions 5-13
sysreg.h – Register Access 5-14
trans.h – Fast Fourier Transforms 5-14
vector.h – Vector Functions 5-14

window.h – Window Generators 5-14
Built-In DSP Functions 5-15
Implications of Using SIMD Mode 5-17
DSP Run-Time Library Reference 5-19
a_compress 5-20
a_expand 5-21
alog 5-22
alog10 5-23
arg 5-24
autocoh 5-25
autocorr 5-27
biquad 5-29
cabs 5-32
cadd 5-33
cartesian 5-34
cdiv 5-36
cexp 5-37
cfft_mag 5-38
cfftN 5-40
cfftf 5-43
cmatmadd 5-46
cmatmmlt 5-48
cmatmsub 5-50
cmatsadd 5-52

cmatsmlt 5-54
cmatssub 5-56
cmlt 5-58
conj 5-59
convolve 5-60
copysign 5-62
cot 5-63
crosscoh 5-64
crosscorr 5-66
csub 5-68
cvecdot 5-69
cvecsadd 5-71
cvecsmlt 5-73
cvecssub 5-75
cvecvadd 5-77
cvecvmlt 5-79
cvecvsub 5-81
favg 5-83
fclip 5-84
fir 5-85
fmax 5-87
fmin 5-88
gen_bartlett 5-89
gen_blackman 5-91

gen_gaussian 5-92
gen_hamming 5-93
gen_hanning 5-94
gen_harris 5-95
gen_kaiser 5-96
gen_rectangular 5-98
gen_triangle 5-99
gen_vonhann 5-101
histogram 5-102
idle 5-104
ifftN 5-105
iir 5-108
matinv 5-112
matmadd 5-113
matmmlt 5-115
matmsub 5-117
matsadd 5-119
matsmlt 5-121
matssub 5-123
mean 5-125
mu_compress 5-126
mu_expand 5-127
norm 5-128
polar 5-129

poll_flag_in 5-131
rfft_mag 5-133
rfftN 5-135
rms 5-138
rsqrt 5-139
set_flag 5-140
set_semaphore 5-142
timer_off 5-143
timer_on 5-144
timer_set 5-145
transpm 5-147
twidfftf 5-149
var 5-151
vecdot 5-153
vecsadd 5-155
vecsmlt 5-157
vecssub 5-159
vecvadd 5-161
vecvmlt 5-163
vecvsub 5-165
zero_cross 5-167

INDEX

PREFACE

Thank you for purchasing Analog Devices, Inc. development software for signal processing applications.

Purpose of This Manual

The *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for SHARC Processors* contains information about the C/C++ compiler and run-time library for SHARC® (ADSP-21xxx) 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 SHARC architecture and instruction set and the C/C++ programming languages.

Programmers who are unfamiliar with SHARC processors can use this manual, but they should supplement it with other texts (such as the appropriate hardware reference and programming reference manuals) that describe your target architecture.

Manual Contents

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"
  Shows how to optimize compiler operation
- Chapter 3, "C/C++ Run-Time Library"
  Shows how to use library functions and provides a complete C/C++ library function reference (for functions covered in the current compiler release)
- Chapter 4, "DSP Library for ADSP-2106x and ADSP-21020 Processors"
  Shows how to use DSP library functions and provides a complete library function reference used with ADSP-2106x and ADSP-21020 processors (for functions covered in the current compiler release)
- Chapter 5, "DSP Library for ADSP-2116x/2126x/2136x Processors"
  Shows how to use DSP library functions and provides a complete library function reference used with ADSP-2116x/2126x/2136x processors (for functions covered in the current compiler release)

What's New in This Manual

This edition of the *VisualDSP++ 4.0 C/C++ Compiler and Library Manual for SHARC Processors* documents support for all current SHARC processors listed in "Supported Processors".

Refer to the *VisualDSP++ 4.0 Product Release Bulletin* 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 “SHARC” refers to a family of Analog Devices, Inc. high-performance 32-bit floating-point digital signal processors that can be used in speech, sound, graphics, and imaging applications. VisualDSP++® currently supports the following SHARC processors:

ADSP-21020	ADSP-21060	ADSP-21061	ADSP-21062
ADSP-21065L	ADSP-21160	ADSP-21161	ADSP-21261
ADSP-21262	ADSP-21266	ADSP-21267	ADSP-21363
ADSP-21364	ADSP-21365	ADSP-21366	ADSP-21367
ADSP-21368	ADSP-21369		

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 notifications containing updates to the Web pages 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 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 or ftp 137.71.23.21
  ftp://ftp.analog.com

Related Documents

For information on product related development software, see these publications:

- *VisualDSP++ 4.0 Getting Started Guide*
- *VisualDSP++ 4.0 User's Guide*
- *VisualDSP++ 4.0 C/C++ Assembler and Preprocessor Manual*
- *VisualDSP++ 4.0 Linker and Utilities Manual*
- *VisualDSP++ 4.0 Loader Manual*
- *VisualDSP++ 4.0 Product Release Bulletin*
- *VisualDSP++ Kernel (VDK) User's Guide*
- *Quick Installation Reference Card*

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

Visit the Technical Library Web site to access all processor and tools manuals and data sheets:

```
http://www.analog.com/processors/resources/technicalLibrary
```

Online Technical Documentation

Online documentation includes 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 using the Search function of VisualDSP++ Help system. For easy printing, supplementary `.PDF` files of most manuals are also provided.

Each documentation file type is described as follows.

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 requires a browser, such as Internet Explorer 4.0 (or higher).
`.PDF`	VisualDSP++ and processor manuals in Portable Documentation Format (PDF). Viewing and printing the `.PDF` files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

Access the online documentation from the VisualDSP++ environment, Windows® Explorer, or the Analog Devices Web site.

Accessing Documentation From VisualDSP++

From the 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).

Accessing Documentation 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 of VisualDSP++ environment. The `.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 the FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click the `vdsp-help.chm` file, which is the master Help system, to access all the other `.CHM` files.
- Open your VisualDSP++ installation CD-ROM and 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**.

Accessing Documentation From the Web

Download manuals in PDF format 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. Only production (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**); they also can be downloaded from the Web site.

To have a data sheet faxed to you, call the Analog Devices Faxback System at **1-800-446-6212**. Follow the prompts and a list of data sheet code numbers will be faxed to you. If the data sheet you want is not listed, check for it on the Web site.

Notation Conventions

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

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

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

Example	Description
	Note: For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
	Caution: Incorrect device operation may result if ... **Caution:** Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word **Caution** appears instead of this symbol.
	Warning: Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word **Warning** appears instead of this symbol.

1 COMPILER

The C/C++ compiler (`cc21k`) is part of Analog Devices development software for SHARC (ADSP-21xxx) processors.

The code examples in this manual have been compiled using VisualDSP++ 4.0. The examples compiled with other versions of VisualDSP++ may result in build errors or different output although the highlighted algorithms stand and should continue to stand in future releases of VisualDSP++.

This chapter contains:

- "C/C++ Compiler Overview" on page 1-2
  provides an overview of C/C++ compiler for SHARC 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 1-70
  describes the `cc21k` compiler's extensions to the ISO/ANSI standard for the C and C++ languages.
- "Preprocessor Features" on page 1-165
  contains information on the preprocessor and ways to modify source compilation.
- "C/C++ Run-Time Model and Environment" on page 1-175
  contains reference information about implementation of C/C++ programs, data, and function calls in ADSP-21xxx processors.

- "C/C++ and Assembly Interface" on page 1-209
  describes how to call an assembly language subroutine from a C or C++ program, and how to call a C or C++ function from within an assembly language program.
- "Compiler C++ Template Support" on page 1-238
  describes how templates are instantiated at compile time

C/C++ Compiler Overview

The C/C++ compiler (`cc21k`) is designed to aid your 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 processor operations without having to understand the underlying processor architecture.

The C/C++ compiler (`cc21k`) compiles ISO/ANSI standard C and C++ code for the SHARC processors. Additionally, Analog Devices includes within the compiler a number of C language extensions designed to assist in processor development. The compiler runs from the VisualDSP++ environment or from an operating system command line.

The C/C++ compiler (`cc21k`) processes your C and C++ language source files and produces SHARC assembler source files. The assembler source files are assembled by the SHARC assembler (`easm21k`). The assembler creates Executable and Linkable Format (ELF) object files that can either be linked (using the linker) to create an ADSP-21xxx executable file or included in an archive library (`elfar`). The way in which the compiler controls the assemble, link, and archive phases of the process depends on the source input files and the compiler options used.

Your source files contain the C/C++ program to be processed by the compiler. The `cc21k` 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 `cc21k` compiler supports a set of C/C++ language extensions. These extensions support hardware features of the ADSP-21xxx processors. For information on these extensions, see "C/C++ Compiler Language Extensions" on page 1-70.

You can set the compiler options from the **Compile** page of the **Project Options** dialog box of the VisualDSP++ Integrated Development and Debug Environment (IDDE) (see "Specifying Compiler Options in VisualDSP++" on page 1-9). These selections control how the compiler processes your source files, letting you select features that include the language dialect, error reporting, and debugger output.

For more information on the VisualDSP++ environment, see the *VisualDSP++ 4.0 User's Guide* and online Help.

Compiler Command-Line Interface

This section describes how the ADSP-21xxx 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 Type and Data Type Sizes" on page 1-61
- "Environment Variables Used by the Compiler" on page 1-63
- "Optimization Control" on page 1-64

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 1-7 lists valid extensions of source files the compiler operates upon. 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 1-21. The summaries are followed by descriptions of each switch.

When developing a project, sometimes it is useful to modify the compiler's default options settings. The way the compiler's options are set depends on the environment used to run the processor development software. See "Specifying Compiler Options in VisualDSP++" on page 1-9 for more information.

Running the Compiler

Use the following syntax for the `cc21k` command line:

```
cc21k [-switch [-switch …] sourcefile [sourcefile …]]
```

runs the compiler with

`cc21k`	Name of the compiler program for SHARC processors.
`-switch`	Switch (or switches) to process. The compiler has many switches. These switches select the operations and modes for the compiler and other tools. Command-line switches are case sensitive. For example, `-O` is not the same as `-o`.
`sourceFile`	Name of the file to be preprocessed, compiled, assembled, and/or linked

The name of the source file to be processed:

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

If its length exceeds eight characters or contains spaces, enclose it in straight quotes; for example, "`long file name.c`". The `cc21k` compiler uses the file extension to determine what the file contains (Table 1-2 on page 1-8) and what operations to perform upon it (Table 1-1 on page 1-7).

For example, the following command line

```
cc21k -O -proc ADSP-21161 -Wremarks -o program.dxe source.c
```

runs `cc21k` with:

`-proc ADSP-21161`	Specifies compiler instructions unique to the ADSP-21161 processor
`-O`	Specifies optimization for the compiler
`-Wremarks`	Selects extra diagnostic remarks in addition to warning and error messages
`-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 for the C++ mode

```
cc21k -c++ source.cpp
```

runs `cc21k` with:

`-c++`	Specifies all of the source files to be compiled in C++ mode
`source.cpp`	Specifies the C++ language source file to be compiled

The normal function of `cc21k` 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 files listed in Table 1-1 to perform the specified action. If multiple files are specified, each is first processed to produce an object file; then all object files are presented to the linker.

Table 1-1. File Extensions

Extension	Action
.c .cpp .cxx .cc .c++	Source file is compiled, assembled, and linked
.asm, .dsp, or .s	Assembly language source file is assembled and linked
.doj	Object file (from previous assembly) is linked

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, `cc21k` names the output for you. Table 1-2 on page 1-8 lists the type of files, names, and extensions `cc21k` 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 1-34).

When providing 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 `cc21k` 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, `cc21k` 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.

Table 1-2. Input and Output Files

Input File Extension	File Extension Description
`.c`	C source file.
`.cc`, `.cpp`, `.cxx`	C++ source file
`.h`	Header file (referenced by a `#include` directive)
`.hpp .hh .hxx .h++`	C++ header file (referenced by a `#include` statement)
`.pch`	C++ pre-compiled header file
`.ii`, `.ti`	Template instantiation files – used internally by the compiler when instantiating templates
`.ipa`, `.opa`	Interprocedural analysis files – used internally by the compiler when performing interprocedural analysis
`.i`	Preprocessed C source, created when preprocess only is specified
`.s`, `.asm`	Assembler source file
`.is`	Preprocessed assembly source (retained when `-save-temps` is specified)
`.ldf`	Linker Description File
`.doj`	Object file to be linked
`.dlb`	Library of object files to be linked as needed
`.dxe`	Executable file produced by compiler
`.xml`	Processor memory map file output
`.sym`	Processor symbol map file output

The compiler refers to a number of environment variables during its operation, and these environment variables can affect the compiler's behavior. Refer to "Environment Variables Used by the Compiler" on page 1-63 for more information.

Specifying Compiler Options in VisualDSP++

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

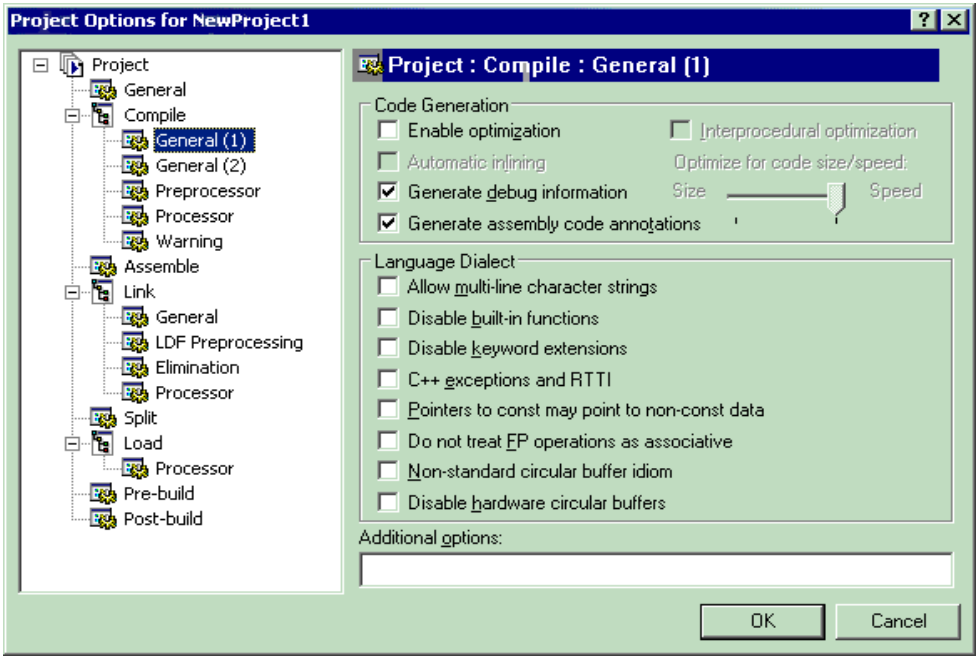


Figure 1-1. Project Options — Compile Tab

There are four sub-pages you can access—**General**, **Preprocessor**, **Processor**, and **Warning**. Most page options have a corresponding compiler command-line switch described in "Compiler Command-Line Switches"

on page 1-11. The **Additional options** field 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.

For example, the **Processor** page (Figure 1-2) provides selection of data word sizes, processor speed, and other options.

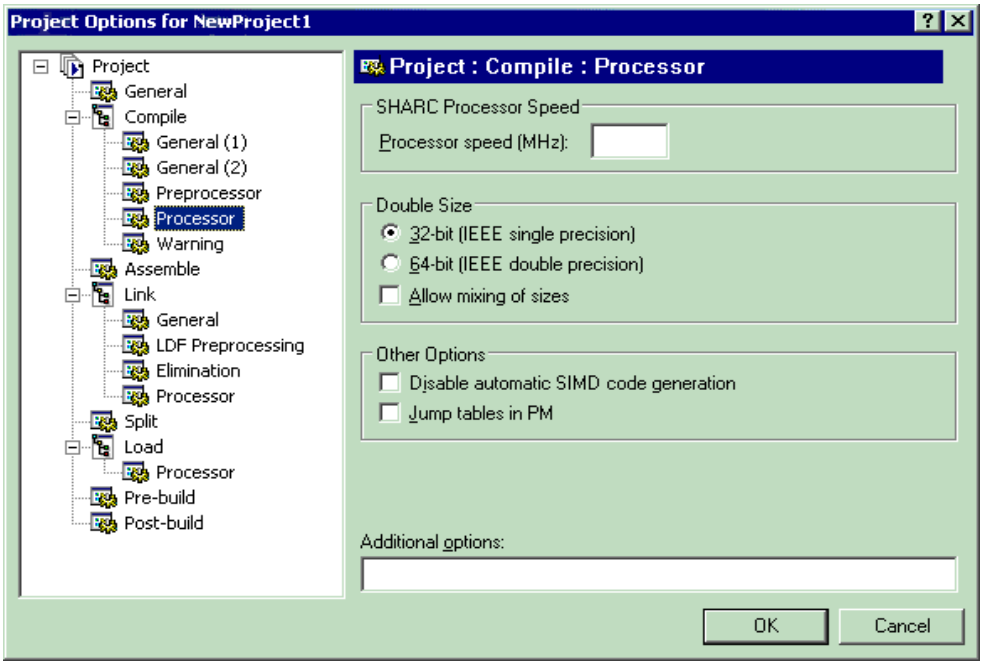


Figure 1-2. Project Options – Compile (Processor) Page

For more information, use the VisualDSP++ context-sensitive online Help for each target architecture to select information on compiler options you can specify in VisualDSP++. To do that, click on the **?** button and then click in a field or box you need information about.

Compiler Command-Line Switches

This section describes the command-line switches used when compiling. It contains a set of tables that provide a brief description of each switch. These tables are organized by type of 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).

- "C or C++ Mode Selection Switches", Table 1-3
- "C/C++ Compiler Common Switches", Table 1-4
- "C++ Mode Compiler Switches", Table 1-5 on page 1-21

A brief description of each switch follows the tables, beginning on page 1-22.

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

Switch Name	Description
`-c89` (on page 1-22)	Supports programs that conform to the ISO/IEC 9899:1990 standard
`-c++` (on page 1-22)	Supports ANSI/ISO standard C++ with Analog Devices extensions. Note that C++ is not supported on the ADSP-21020 processor.

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

Switch Name	Description
`sourcefile` (on page 1-23)	Specifies file to be compiled
`-@ filename` (on page 1-23)	Reads command-line input from the file
`-A name[tokens]` (on page 1-23)	Asserts the specified name as a predicate
`-aligned-stack` (on page 1-24)	Aligns the program stack on a double-word boundary
`-alttok` (on page 1-24)	Allows alternative keywords and sequences in sources
`-annotate-loop-instr` (on page 1-25)	Provides additional annotation information for the prolog, kernel and epilog of a loop
`-bss` (on page 1-25)	Places zero-initialized global data into a BSS section
`-build-lib` (on page 1-25)	Directs the librarian to build a library file
`-C` (on page 1-25)	Retains preprocessor comments in the output file; must run with the `-E` or `-P` switch
`-c` (on page 1-26)	Compiles and/or assembles only, but does not link
`-compatible-pm-dm` (on page 1-26)	Specifies that the compiler shall treat `dm`- and `pm`-qualified pointers as assignment-compatible
`-const-read-write` (on page 1-26)	Specifies that data accessed via a pointer to `const` data may be modified elsewhere
`-Dmacro[=definition]` (on page 1-26)	Defines a macro.
`-debug-types` (on page 1-26)	Supports building an `*.h` file directly and writing a complete set of debugging information for the header file
`-default-linkage-{asm|C|C++}` (on page 1-27)	Sets the default linkage type (`C`, `C++`, `asm`)

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

Switch Name	Description
`-double-size [-32|-64]` (on page 1-27)	Selects 32- or 64-bit IEEE format for `double`. The `-double-size-32` is the default mode.
`-double-size-any` (on page 1-28)	Indicates that the resulting object can be linked with objects built with any `double` size
`-dry` (on page 1-28)	Displays, but does not perform, main driver actions (verbose dry-run)
`-dryrun` (on page 1-28)	Displays, but does not perform, top-level driver actions (terse dry-run)
`-E` (on page 1-29)	Preprocesses, but does not compile, the source file
`-ED` (on page 1-29)	Preprocesses and sends all output to a file
`-EE` (on page 1-29)	Preprocesses and compiles the source file
`-extra-keywords` (on page 1-29)	Recognizes ADI extensions to ANSI/ISO standards for C and C++ (default mode)
`-flags-{tools} <arg1> [,arg2...]` (on page 1-29)	Passes command-line switches through the compiler to other build tools
`-float-to-int` (on page 1-30)	Uses a support library function to convert a `float` to an `integer`
`-force-circbuf` (on page 1-26)	Treats array references of the form `array[i%n]` as circular buffer operations
`-fp-associative` (on page 1-30)	Treats floating-point multiply and addition as an associative
`-full-version` (on page 1-30)	Displays the version number of the driver and any processes invoked by the driver
`-g` (on page 1-31)	Generates DWARF-2 debug information

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

Switch Name	Description
`-H` (on page 1-31)	Outputs a list of included header files, but does not compile
`-HH` (on page 1-31)	Outputs a list of included header files and compiles
`-h[elp]` (on page 1-32)	Outputs a list of command-line switches
`-I` *`directory`* (on page 1-32)	Appends directory to the standard search path
`-I-` (on page 1-32)	Establishes the point in the `include` directory list at which the search for header files enclosed in angle brackets should begin
`-i` (on page 1-32)	Outputs only header details or makefile dependencies for `include` files specified in double quotes
`-include` *`filename`* (on page 1-33)	Includes named file prior to preprocessing each source file
`-ipa` (on page 1-33)	Enables interprocedural analysis
`-L` *`directory`* (on page 1-34)	Appends directory to the standard library search path
`-l` *`library`* (on page 1-34)	Searches library for functions when linking
`-M` (on page 1-35)	Generates make rules only, but does not compile
`-MD` (on page 1-35)	Generates `make` rules, compiles, and prints to a file
`-MM` (on page 1-35)	Generates make rules and compiles
`-Mo` *`filename`* (on page 1-35)	Writes dependency information to *`filename`*. This switch is used in conjunction with the `-ED` or `-MD` options

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

Switch Name	Description
`-Mt filename` (on page 1-35)	Makes dependencies, where the target is renamed as *filename*
`-MQ` (on page 1-36)	Generates make rules only; does not compile. No notification when input files are missing
`-map filename` (on page 1-36)	Directs the linker to generate a memory map of all symbols
`-mem` (on page 1-36)	Enables memory initialization
`-multiline` on page 1-36	Enables string literals over multiple lines (default)
`-no-aligned-stack` (on page 1-36)	Does not double-word align the program stack
`-no-alttok` (on page 1-36)	Does not allow alternative keywords and sequences in sources
`-no-annotate` (on page 1-37)	Disables the annotation of assembly files
`-no-annotate-loop-instr` (on page 1-37)	Disables the production of additional loop annotation information by the compiler (default mode)
`-no-bss` (on page 1-37)	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 1-37)	Recognizes only built-in functions that begin with two underscores(__)
`-no-circbuf` (on page 1-38)	Disables the automatic generation of circular buffer code by the compiler
`-no-db` (on page 1-38)	Specifies that the compiler shall not generate code containing delayed branches jumps
`-no-defs` (on page 1-38)	Disables preprocessor definitions: macros, include directories, library directories, run-time headers, or keyword extensions

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

Switch Name	Description
`-no-extra-keywords` (on page 1-38)	Does not accept ADI keyword extensions that might affect ISO/ANSI standards for C and C++
`-no-fp-associative` (on page 1-38)	Does not treat floating-point multiply and addition as an associative
`-no-mem` (on page 1-39)	Disables memory initialization
`-no-multiline` (on page 1-39)	Disables multiple line string literal support
`-no-saturation` (on page 1-39)	Causes the compiler not to introduce saturation semantics when optimizing expressions
`-no-simd` (on page 1-39)	Disables automatic SIMD mode when compiling for ADSP-2116x, ADSP-2126x or ADSP-2136x processors
`-no-std-ass` (on page 1-39)	Disables any predefined assertions and system-specific macro definitions
`-no-std-def` (on page 1-40)	Disables preprocessor definitions and ADI keyword extensions that do not have leading underscores(__)
`-no-std-inc` (on page 1-40)	Searches for preprocessor include header files only in the current directory and in directories specified with the `-I` switch
`-no-std-lib` (on page 1-40)	Searches for only those library files specified with the `-l` switch
`-no-threads` (on page 1-40)	Specifies that all compiled code need not be thread-safe
`-O [0|1]` (on page 1-40)	Enables code optimizations
`-Oa` (on page 1-41)	Enables automatic function inlining
`-Og` (on page 1-41)	Enables a compiler mode that performs optimizations while still preserving the debugging information

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

Switch Name	Description
`-Os` (on page 1-41)	Optimizes for code size
`-Ov num` (on page 1-41)	Controls speed versus size optimizations
`-o` *`filename`* (on page 1-42)	Specifies the output file name
`-P` (on page 1-42)	Preprocesses, but does not compile, the source file; omits line numbers in the preprocessor output
`-PP` (on page 1-42)	Similar to `-P`, but does not halt compilation after preprocessing
`-path-{asm|compiler|lib|link|mem}` *`directory`* (on page 1-42)	Uses the specified directory as the location of the specified compilation tool (assembler, compiler, librarian, linker, or and memory initializer, respectively)
`-path-install` *`directory`* (on page 1-42)	Uses the specified directory as the location of all compilation tools
`-path-output` *`directory`* (on page 1-43)	Specifies the location of non-temporary files
`-path-temp` *`directory`* (on page 1-43)	Specifies the location of temporary files
`-pch` (on page 1-43)	Generates and uses precompiled header files (`*.pch`)
`-pchdir` *`directory`* (on page 1-43)	Specifies the location of PCHRepository
`-pedantic` (on page 1-43)	Issues compiler warnings for any constructs that are not ISO/ANSI standard C/C++-compliant
`-pedantic-errors` (on page 1-44)	Issues compiler errors for any constructs that are not ISO/ANSI standard C/C++-compliant.
`-pguide` (on page 1-44)	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization

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

Switch Name	Description
`-pplist` *`filename`* (on page 1-44)	Outputs a raw preprocessed listing to the specified file.
`-proc` *`processor`* (on page 1-45)	Specifies that the compiler should produce code suitable for the specified processor
`-progress-rep-func` (on page 1-45)	Issues a diagnostic message each time the compiler starts compiling a new function. Equivalent to `-Wwarn=cc1472`.
`-progress-rep-gen-opt` (on page 1-45)	Issues a diagnostic message each time the compiler starts a new generic optimization pass on the current function. Equivalent to `-Wwarn=cc1473`.
`-progress-rep-mc-opt` (on page 1-46)	Issues a diagnostic message each time the compiler starts a new machine-specific optimization pass on the current function. Equivalent to `-Wwarn=cc1473`.
`-R` *`directory`* (on page 1-46)	Appends directory to the standard search path for source files
`-R-` (on page 1-47)	Removes all directories from the standard search path for source files
`-reserve <reg1>[,reg2...]` (on page 1-47)	Reserves certain registers from compiler use. **Note:** Reserving registers can have a detrimental effect on the compiler's optimization capabilities.
`-restrict-hardware-loops <maximum>` (on page 1-47)	Restrict the number of levels of loop nesting used by the compiler
`-S` (on page 1-47)	Stops compilation before running the assembler
`-s` (on page 1-48)	Removes debug info from the output executable file
`-save-temps` (on page 1-48)	Saves intermediate files
`-section <id>=<section_name>` (on page 1-48)	Orders the compiler to place data/program of type "`id`" into the section "`section_name`"
`-show` (on page 1-48)	Displays the driver command-line information

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

Switch Name	Description
`-si-revision` *`version`* (on page 1-49)	Specifies a silicon revision of the specified processor. The default setting is the latest silicon revision.
`-signed-bitfield` (on page 1-50)	Makes the default type for `int` bitfields signed
`-signed-char` (on page 1-50)	Makes the default type for `char` signed
`-structs-do-not-overlap` (on page 1-51)	Specifies that `struct` copies may use "`memcpy`" semantics, rather than the usual "`memmove`" behavior
`-switch-pm` (on page 1-51)	Specifies that the compiler should place switch tables in program memory
`-syntax-only` (on page 1-51)	Checks the source code for compiler syntax errors, but does not write any output
`-sysdefs` (on page 1-52)	Defines the system definition macros
`-T` *`filename`* (on page 1-52)	Specifies the Linker Description File
`-threads` (on page 1-52)	Specifies that support for multithreaded applications is to be enabled
`-time` (on page 1-53)	Displays the elapsed time as part of the output information on each part of the compilation process
`-U`*`macro`* on page 1-53	Undefines macro(s)
`-unsigned-bitfield` (on page 1-53)	Makes the default type for plain `int` bitfields unsigned
`-unsigned-char` (on page 1-54)	Makes the default type for `char` unsigned
`-v` (on page 1-54)	Displays both the version and command-line information
`-verbose` (on page 1-54)	Displays command-line information

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

Switch Name	Description
`-version` (on page 1-54)	Displays version information
`-W{error|remark|suppress|warn} number` (on page 1-55)	Overrides the default severity of the specified error message
`-Wdriver-limit number` (on page 1-55)	Aborts the driver after reaching the specified number of errors
`-Werror-limit number` (on page 1-55)	Stops compiling after reaching the specified number of errors
`-Wremarks` (on page 1-55)	Indicates that the compiler may issue remarks, which are diagnostic messages even milder than warnings
`-Wterse` (on page 1-55)	Issues only the briefest form of compiler warning, errors, and remarks.
`-w` (on page 1-56)	Does not display compiler warning messages
`-warn-protos` (on page 1-56)	Produces a warnings when a function is called without a prototype
`-workaround <workaround>` (on page 1-56)	Enables code generator workaround for specific hardware errata
`-write-files` (on page 1-56)	Enables compiler I/O redirection
`-write-opts` (on page 1-57)	Passes the user options (but not input filenames) via a temporary file
`-xref filename` (on page 1-57)	Outputs cross-reference information to the specified file

Table 1-5. C++ Mode Compiler Switches

Switch Name	Description
`-anach` on page 1-58	Supports some language features (anachronisms) that are prohibited by the C++ standard but still in common use
`-eh` on page 1-59	Enables exception handling
`-no-anach` on page 1-60	Disallows the use of anachronisms that are prohibited by the C++ standard
`-no-demangle` (on page 1-60)	Prevents filtering of any linker errors through the demangler
`-no-eh` on page 1-60	Disables exception-handling
`-no-implicit-inclusion` (on page 1-60)	Prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated
`-no-rtti` on page 1-60	Disables run-time type information
`-rtti` on page 1-61	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 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` is 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).

C++ is not supported on the ADSP-21020 processor.

C/C++ Compiler Common Switch Descriptions

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

sourcefile

The *sourcefile* parameter (or parameters) specifies the name of the file (or files) to be preprocessed, compiled, assembled, and/or linked. A file name can include the drive, directory, file name, and file extension. The `cc21k` 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*.

-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	`adsp21xxx`
cpu	`adsp21xxx`
compiler	`cc21k`

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

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

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

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

For example, the default assertions may be tested as:

```
#if #machine(adsp21xxx)
    // do something
#endif
```

The parentheses in the assertion need quotes when using the `-A` switch, to prevent misinterpretation. No quotes are needed for a `#assert` directive in a source file.

-aligned-stack

The `-aligned-stack` switch directs the compiler to align the program stack on a double-word boundary.

-alttok

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

Keyword	Equivalent
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
or	||
or_eq	|=
not	!

Keyword	Equivalent
not_eq	!=
xor	^
xor_eq	^=

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

-annotate-loop-instr

The `-annotate-loop-instr` switch directs the compiler to provide additional annotation information for the prolog, kernel and epilog of a loop. See "Assembly Optimizer Annotations" on page 2-51 for more details on this feature.

-bss

The `-bss` (build library) switch directs the compiler to place global data into a BSS-style section (called "bsz"), rather than into normal global data section. See also "-no-bss" on page 1-37.

-build-lib

The `-build-lib` (build library) switch directs the compiler to use the 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 may only be run in combination with the `-E` or `-P` switches, directs the C/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.

-compatible-pm-dm

The `compatible-pm-dm` switch specifies that the compiler shall treat `dm`- and `pm`-qualified pointers as assignment-compatible.

-const-read-write

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

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

-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'. Note that the compiler processes all `-D` switches on the command line before any `-U` (undefine macro) switches.

-debug-types

The `-debug-types` switch builds a `*.h` file directly and writes 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,

```
cc21k -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-linkage[**-asm | -C | -C++**]

The `-default-linkage-asm` (assembler linkage), `-default-linkage-C` (C linkage), and `-default-linkage-C++` (C++ linkage) 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.

Invoke this switch with the **Additional Options** box located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-double-size[**-32|-64**]

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

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 inadvertently use slow speed emulated 64-bit floating-point arithmetic, even when variables are declared consistently as `float`.

To avoid this problem, `cc21k` 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 `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 standard requires 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.

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__` macro, while the `-double-size-64` switch undefines the `__DOUBLES_ARE_FLOATS__` macro.

-double-size-any

The `-double-size-any` switch specifies that the resulting object files should be marked in such a way that will enable them to be linked against objects built with `doubles` either 32-bit or 64-bit in size.

-dry

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

-dryrun

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

-E

The `-E` (stop after preprocessing) switch directs the compiler to stop after the C/C++ preprocessor runs (without compiling). The output (preprocessed source code) prints to the standard output stream (`<stdout>`) unless the output file is specified with the `-o` switch. Note that the `-C` switch can only be run in combination with the `-E` switch.

Invoke this switch 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 ANSI/ISO standard C and C++, such as `pm` and `dm`, without leading underscores, which can affect conforming ANSI/ISO C and C++ programs. This is the default mode.

-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. The tools are:

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

-float-to-int

The `-float-to-int` switch instructs the compiler to use a support library function to convert a `float` to an integer. The library support routine performs extra checking to avoid a floating-point underflow occurring.

-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 conservative, and does not use circular buffers unless it can verify that the circular index or pointer is always within the circular buffer range. See "Circular Buffer Built-In Functions" on page 1-107.

-fp-associative

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

-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 enable optimization (`-O`) switch, the compiler performs standard optimizations. The compiler also outputs symbols and other information to provide limited source-level debugging through the VisualDSP++ IDDE (debugger). This combination of options provides line debugging and global variable debugging.

When the `-g` and `-O` switches are specified, no debug information is available for local variables and the standard optimizations can sometimes rearrange program code in a way that inaccurate line number information may be produced. For full debugging capabilities, use the `-g` switch without the `-O` switch. See also the `-Og` switch (on page 1-41).

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 output stream 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 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 can be specified more than once; all specified directories are added to the search path.

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

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

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

-I-

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

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

-i

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

-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 turns on Interprocedural Analysis (IPA) in the compiler. This option enables optimization across the entire program, including between source files that were compiled separately. The `-ipa` option should be applied to all C and C++ files in the program. For more information, see “Interprocedural Analysis” on page 1-67. Specifying `-ipa` also implies setting the `-O` switch (on page 1-40).

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 and global variables when linking. The library name is the portion of the file name between the `lib` prefix and `.dlb` extension.

For example, the `-lc` compiler switch directs the linker to search in the library named `c`. This library resides in a file named `libc.dlb`.

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

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

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

-M

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

```
object-file: include-file …
```

-MD

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

-MM

The `-MM` (generate make rules and compile) switch directs the preprocessor to print to standard out 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 modifies the target of dependencies, renaming the target to filename. It only has an effect when used in conjunction with the `-M` or `-MM` switch..

-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` *`filename`* (generate a memory map) switch directs the linker to output a memory map of all symbols. The map file name corresponds to the *`filename`* argument. For example, if the 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 run the `mem21k` initializer.

-multiline

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

-no-aligned-stack

The `-no-aligned-stack` (disable stack alignment) switch directs the compiler to not align the program stack on a double-word boundary.

-no-alttok

The `-no-alttok` (disable alternative tokens) switch directs the compiler not to accept alternative operator keywords and digraph sequences in the source files. This is the default mode. For more information, see "-alttok" on page 1-24.

-no-annotate

The `-no-annotate` (disable assembly annotations) switch directs the compiler not to annotate assembly files. The default behavior 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.

-no-annotate-loop-instr

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

-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 1-25.

-no-builtin

The `-no-builtin` (no built-in functions) switch directs the compiler to ignore built-in functions that begin with two underscores (__). Note that this switch influences many functions. This switch also predefines the `__NO_BUILTIN` preprocessor macro. For more information on built-in functions, see "C++ Fractional Type Support" on page 1-146.

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) are not affected.

-no-db

The `-no-db` (no delayed branches) switch specifies that the compiler shall not generate jumps that use delayed branches.

-no-defs

The `-no-defs` (disable defaults) switch directs the preprocessor not to define any default preprocessor macros, include directories, library directories, libraries, and run-time headers. It also disables the Analog Devices `cc21k` 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 standards for C and C++ languages. These include keywords such as `pm` and `dm`, which may be used as identifiers in standard conforming programs. Alternate keywords, which are prefixed with two leading underscores, such as `__pm` and `__dm`, continue to work.

-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 `mem21k` initializer. Note that if you use `-no-mem`, the compiler does not initialize globals and statics.

-no-multiline

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

-no-saturation

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

-no-simd

The `-no-simd` (disable SIMD mode) switch directs the compiler to disable automatic SIMD code generation when compiling for ADSP-2116x, ADSP-2126x and ADSP-2136x processors. Note that SIMD code is still generated for a loop if it is preceded with the "`SIMD_for`" pragma. The pragma is treated as an explicit user request to generate SIMD code and is always obeyed, if possible. See "SIMD Support" on page 1-150 for more information.

-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 1-23) for the list of standard assertions.

-no-std-def

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

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

-no-std-inc

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

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

-no-std-lib

The `-no-std-lib` (disable standard library search) switch directs the linker to search for libraries in only the current project directory and directories specified with the `-L` switch.

-no-threads

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

-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 `cc21k` 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` (optimize for speed versus size) switch (on page 1-41). Therefore, use of `-Ov100` indicates that as many functions as possible are auto-inlined, whereas `-Ov0` prevents any function from being auto-inlined. Specifying `-Oa` also implies the use of `-O`.

-Og

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

-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.

-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/C++ preprocessor runs (without compiling) and to omit the `#line` preprocessor command with line number information from the preprocessor output. The `-C` switch can be used in conjunction with `-P` to retain comments.

-PP

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

-path-{ asm | compiler | lib | link | mem } ***directory***

The `-path-{asm|compiler|lib|link|mem}` *`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 comprises a relative or absolute path to its location. Respectively, the tools are the assembler, compiler, librarian, linker or memory initializer. Use this switch when overriding the normal version of one or more of the tools. The `-path-{}` switch also overrides the directory specified by the `-path-install` switch.

-path-install directory

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

You can selectively override this switch with the `-path-tool` switch.

-path-output directory

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

-path-temp directory

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

-pch

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

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

-pchdir directory

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

-pedantic

The `-pedantic` (ANSI standard warnings) switch causes the compiler to issue a warning for each construct found in your program that does not strictly conform to ANSI/ISO standard C or C++.

The compiler may not detect all such 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 standard errors) switch causes the compiler to issue errors instead of warnings for cases described in the `-pedantic` switch.

-pguide

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

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 `#if` or `#ifdef`
`L`	Change in source position
`R`	Diagnostic message (remark)
`W`	Diagnostic message (warning)
`E`	Diagnostic message (error)
`C`	Diagnostic message (catastrophic error)

-proc processor

The `-proc processor` (target processor) switch specifies the compiler produces code suitable for the specified processor. Refer to "Supported Processors" for the list of supported SHARC processors.

For example,

```
cc21k -proc ADSP-21161 -o bin\p1.doj p1.asm
```

If no target is specified with the `-proc` switch, the system uses the `ADSP-21060` processor settings as a default.

When compiling with the `-proc` switch, the appropriate processor macro as well as `__ADSP21000__` are defined as 1. For example, `__ADSP21060__` and `__ADSP-21000__` are 1.

See also "-si-revision version" on page 1-49 for more information on silicon revision of the specified processor.

-progress-rep-func

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

-progress-rep-gen-opt

The `-progress-rep-gen-opt` switch provides feedback on the compiler's progress that may be useful when compiling and optimizing a very large, complex function. It issues a "warning" message each time the compiler starts a new generic optimization pass on the current function. The "warn-

ing" message is a remark that is disabled by default, and this switch enables the remark as a warning. The switch is equivalent to `-Wwarn=cc1473`.

-progress-rep-mc-opt

The `-progress-rep-mc-opt` switch provides feedback on the compiler's progress that may be useful when compiling and optimizing a very large, complex function. It issues a "warning" message each time the compiler starts a new machine-specific optimization pass on the current function. The "warning" message is a remark that is disabled by default, and this switch enables the remark as a warning. The switch is equivalent to `-Wwarn=cc1474`.

-R directory[{:|,}**directory …**]

The `-R` *`directory`* (add source directory) switch directs the compiler to add the specified directory to the list of directories searched for source files. 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. The `-R directory` option is position-dependent on the command line. That is, it affects only source files that follow the option.

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

-R-

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

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

-reserve register[, register …]

The `-reserve` (reserve register) switch directs the compiler not to use the specified registers. This guarantees that a known set of registers are available for inline assembly code or linked assembly modules. Separate each register name with a comma on the compiler command line.

You can reserve the following registers: `b0`, `l0`, `m0`, `i0`, `b1`, `l1`, `m1`, `i1`, `b8`, `l8`, `m8`, `i8`, `b9`, `l9`, `m9`, `i9`, `ustat1`, and `ustat2` (as well as `ustat3` and `ustat4` on ADSP-2116x, ADSP-2126x and ADSP-2136x processors). When reserving an `L` (length) register, you must reserve the corresponding `I` (index) register; reserving an `L` register without reserving the corresponding `I` register may result in execution problems.

-restrict-hardware-loops <maximum>

The `-restrict-hardware-loops <maximum>` switch restricts the level of nested hardware loops that the compiler generates. The default setting is 6, which is the maximum number of levels that the hardware supports.

-S

The `-S` (stop after compilation) switch directs `cc21k` to stop compilation before running the assembler. The compiler outputs an assembly file with an `.s` extension.

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 debug information) switch directs the compiler to remove debug 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`, `*.doj`) files in the temp subdirectory of the current project directory. See Table 1-2 on page 1-8 for a list of intermediate files.

-section <id>=<section_name>[,<id>=section_name>]

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

The compiler currently supports the following section identifiers:

`vtbl` — controls placement of the C++ virtual lookup tables
`switch` — controls placement of tables used in the implementation of switch statements

Make sure that the section selected via the command line exists within the `.LDF` file (refer to the *Linker* chapter in the VisualDSP++ 4.0 Linker and Utilities manual).

-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) switch directs the compiler to build for a specific hardware revision. Any errata workarounds available for the targeted silicon revision will be enabled. The parameter "*version*" represents a silicon revision of the processor specified by the `-proc` switch (on page 1-45). The "`none`" revision disables support for silicon errata.

For example, `cc21k -proc ADSP-21161 -si-revision 0.1 proc.c`

If silicon version "`none`" is used, then no errata workarounds are enabled, whereas specifying silicon version "`any`" will enable all errata workarounds for the target processor.

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

Run-time libraries built without any errata workarounds are located in the platform's `lib` sub-directory; for example, `212xx\lib`. Within the `lib` sub-directory, there are library directories for each silicon revision; these libraries have been built with errata workarounds appropriate for the silicon revision enabled. Note that an individual set of libraries may cover more than one specific silicon revision, so if several silicon revisions are affected by the same errata, then one common set of libraries might be used.

The `__SILICON_REVISION__` macro is set by the compiler to two hexadecimal digits representing the major and minor numbers in the silicon revision. For example, `1.0` becomes `0x100` and `10.21` becomes `0xa15`.

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

The compiler driver will pass the `-si-revision <silicon version>` switch when invoking another VisualDSP++ tool, for example when the compiler driver invokes the assembler and linker.

On ADSP-2116x processors, the workaround for the shadow write FIFO anomaly is only required under certain circumstances and is therefore not enabled by default. If this workaround is required, the "`-workaround swfa`" switch should be used. On the ADSP-21161 processor, anomaly #45 can only occur if the code is executed from external memory, and accordingly the workaround for this anomaly is not enabled by default. If this workaround is required, the "`-workaround 21161-anomaly-45`" switch should be used.

Use `http://www.analog.com/processors/technicalSupport/hardwareAnomalies.html` to get more information on specific anomalies (including anomaly IDs).

-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 affect plain one-bit bitfields which are always unsigned. This is the default mode. See also the `-unsigned-bitfield` switch (on page 1-53).

-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.

-structs-do-not-overlap

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

For example, in the statement

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

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

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

Invoke this switch from a check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-switch-pm

The `-switch-pm` (switch tables) switch directs the compiler to place switch tables in program memory.

-syntax-only

The `-syntax-only` (check syntax only) switch directs the compiler to check the source code for syntax errors but not to 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 are defined if the system returns information for them.

Macro	Description
`__HOSTNAME__`	The name of the host machine
`__MACHINE__`	The type of the host machine
`__SYSTEM__`	The Operating System name of the host machine
`__USERNAME__`	The current user's login name
`__GROUPNAME__`	The current user's group name
`__REALNAME__`	The current user's real name

Note that the `__MACHINE__`, `__GROUPNAME__`, and `__REALNAME__` macros are not available on Windows platforms.

-T filename

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

-threads

When used, the `-threads` (enable thread-safe build) switch defines the macro `__ADI_THREADS` as one (`1`) at the compile, assemble and link phases of a build. This specifies that certain aspects of the build are to be done in a thread-safe way.

The use of thread-safe libraries is necessary in conjunction with the `-threads` flag when using the VisualDSP++ Kernel (VDK). The thread-safe libraries can be used with other RTOSs but this requires the definition of various VDK interfaces.

-time

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

-Umacro

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

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 affect 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 1-50).

-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.

-verbose

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

-version

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

-W[error|remark|suppress|warn] number[,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 *`number`* 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.) at which the driver aborts.

-Werror-limit number

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

-Wremarks

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

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

-Wterse

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

-w

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

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. Valid "workarounds" are:

Workaround	Description
rframe	Specifies that the compiler shall not generate the `rframe` instruction as part of the function return sequence.
21161-anomaly-45	Specifies that the compiler does not generate conditional `DAG1` instructions.
swfa	Specifies that the compiler does not generate code that could be affected by the shadow write FIFO anomaly on ADSP-2116x chips

-write-files

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

-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.

The `symbol-id` identifier represents a unique decimal number for the symbol, and `ref-code` is a character from one the following:

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 the C++ compiler.

-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. This is allowed only if anachronisms are enabled and the assignment to this configuration parameter is enabled.
- 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 unnested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-`const` type may be initialized from a value of a different type. A temporary is created; it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-`const` class type may be initialized from an `rvalue` of the `class` type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following statements declare the overload of two functions named `f`.

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

-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 exceptions. However, exceptions can only be thrown from and caught, and cleanup code executed, in modules compiled with `-eh`.

-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 1-58) 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 1-59) for more information.

-no-implicit-inclusion

The `-no-implicit-inclusion` switch prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.

-no-rtti

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

-rtti

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

Data Type 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.

Table 1-6. Data Type Sizes for the ADSP-21xxx Processors

Type	Bit Size	Result of sizeof operator
`int`	32 bits signed	1
`unsigned int`	32 bits unsigned	1
`long`	32 bits signed	1
`unsigned long`	32 bits unsigned	1
`char`	32 bits signed	1
`unsigned char`	32 bits unsigned	1
`short`	32 bits signed	1
`unsigned short`	32 bits unsigned	1
`pointer`	32 bits	1
`float`	32 bits float	1
`fract`	32 bits fixed-point	1
`double`	either 32 or 64 bits float (default 32)	either 1 or 2 (default 1)
`long double`	64 bits float	2

Analog Devices does not support data sizes smaller than the addressable unit size on the processor. For the ADSP-21xxx processors, this means that both `short` and `char` have the same size as `int`. Although 32-bit `chars` are unusual, they do conform to the standard. For information about the `fract` data type, refer to "C++ Fractional Type Support" on page 1-146.

Integer Data Types

On any platform, the basic type `int` is the native word size. For SHARC processors, it is 32 bits. Many library functions are available for 32-bit integers, and these functions provide support for the C/C++ data types `int` and `long int`. Pointers are the same size as `int`s. The `long long int` data type is not supported.

Floating-Point Data Types

For SHARC processors, the `float` data type is 32 bits long. The `double` data type is option-selectable for 32 or 64 bits. The C and C++ languages tend 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 Devices provides the "-double-size[-32|-64]" switch (see 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, the `long double` data type 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 */
```

For full descriptions of these functions and their implementation, see Chapter 3, "C/C++ Run-Time Library".

Environment Variables Used by the Compiler

The compiler refers to a number of environment variables during its operation, as listed below. The majority of the environment variables identify *path names* to directories. You should be aware that placing network paths into these environment variables may adversely affect the time required to compile applications.

- **PATH**
  This is your System search path, used to locate Windows applications when you run them. Windows uses this environment variable to locate the compiler when you execute it from the command line.
- **TMP**
  This directory is used by the compiler for temporary files, when building applications. For example, if you compile a C file to an object file, the compiler first compiles the C file to an assembly file which can be assembled to create the object file. The compiler usually creates a temporary directory within the TMP directory into which to put such files. However, if the `-save-temps` switch is specified, the compiler creates temporary files in the current directory instead. This directory should exist and be writable. If this directory does not exist, the compiler issues a warning.

- **TEMP**
  This environment variable is also used by the compiler when looking for temporary files, but only if TMP was examined and was not set or the directory that TMP specified did not exist.
- **ADI_DSP**
  The compiler locates other tools in the tool-chain through the VisualDSP++ installation directory, or through the `-path-install` switch. If neither is successful, the compiler looks in `ADI_DSP` for other tools.
- **CC21K_OPTIONS**
  If this environment variable is set, and `CC21K_IGNORE_ENV` is not set, this environment variable is interpreted as a list of additional switches to be appended to the command line. Multiple switches are separated by spaces or new lines. A vertical-bar (|) character may be used to indicate that any switches following it will be processed after all other command-line switches.
- **CC21K_IGNORE_ENV**
  If this environment variable is set, `CC21K_OPTIONS` is ignored.

Optimization Control

The general aim of compiler optimization 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. Therefore, the compiler optimizer has a number of configurations, or optimization levels, which can be applied when needed. Each of these levels are enabled by one or more compiler switches (and VisualDSP++ project options) or pragmas.

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

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 1-31 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 1-119).
- **Procedural Optimizations**
  The compiler performs advanced, aggressive optimization on each procedure in the file being compiled. The optimizations can be directed to favor optimizations for speed (`-O1` or `-O`) or space (`-Os`) or a factor between speed and space (`-Ov`). If debugging is also requested, the optimization is given priority so the debugging functionality may be limited. See "-O[0|1]" on page 1-40, "-Os" on page 1-41, and "-Ov num" on page 1-41. Procedural optimizations for speed and space (`-O` and `-Os`) can be enabled in C/C++ source using the pragma `optimize_{for_speed|for_space}` (see on page 1-119 for more information on optimization pragmas).
- **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 1-44 for more information.

The most common scenario in collecting PGO data is 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 a single `.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 submenu added to the top-level **Tools** menu: **Tools -> PGO -> Manage Data Sets**. For more information on PGO, see the *VisualDSP++ 4.0 Getting Started Guide* 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 reduces 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 1-41, "-Ov num" on page 1-41, and "-O[0|1]" on page 1-40 for more information.

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

- **Interprocedural Optimizations**
  The compiler performs advanced, aggressive optimization over the whole program, in addition to the per-file optimizations in procedural optimization. The *interprocedural analysis* (IPA) is enabled using the `-ipa` switch and additionally enables procedural optimizations (`-O`). See "Interprocedural Analysis", "-ipa" on page 1-33 and "-O[0|1]" on page 1-40 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 1-116).

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 `cc21k` compiler has a capability called *interprocedural analysis* (IPA), an optimization 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 1-119).

Use of the `-ipa` switch generates additional files 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, do not use the `-S` option to see the final optimized assembler file when `-ipa` is enabled. Instead, use the `-save-temps` switch (on page 1-48), 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.

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 unnecessary. 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 does 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:

- "Inline Function Support Keyword (inline)" on page 1-74
- "Inline Assembly Language Support Keyword (asm)" on page 1-75
- "Dual Memory Support Keywords (pm dm)" on page 1-90
- "Placement Support Keyword (section)" on page 1-95
- "Placement of Compiler-Generated Data" on page 1-95
- "Boolean Type Support Keywords (bool, true, false)" on page 1-96
- "Pointer Class Support Keyword (restrict)" on page 1-96
- "Variable-Length Array Support" on page 1-97
- "Non-Constant Initializer Support" on page 1-99
- "Indexed Initializer Support" on page 1-99
- "Aggregate Constructor Expression Support" on page 1-101
- "Preprocessor Generated Warnings" on page 1-102
- "C++ Style Comments" on page 1-102
- "Compiler Built-In Functions" on page 1-102
- "Pragmas" on page 1-110
- "GCC Compatibility Extensions" on page 1-138
- "C++ Fractional Type Support" on page 1-146
- "Saturated Arithmetic" on page 1-149
- "SIMD Support" on page 1-150

The additional keywords that are part of the C/C++ extensions do not conflict with any 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 (on page 1-38) is used, the compiler defines the shorter form of the keyword extension that omits the leading underscores.

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 interchange `inline` and `__inline` in your code.

You might need to use the longer form (such as `__inline`) exclusively if 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, 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.

This section provides an overview of the extensions, with brief descriptions, and directs you to text with more information on each extension.

Table 1-7 provides a brief description of each keyword extension and directs you to sections of this chapter that document the extensions in more detail. Table 1-8 provides a brief description of each operational extension and directs you to sections that document these extensions in more detail.

Table 1-7. Keyword Extensions

Keyword extensions	Description
`inline(`function)	Directs the compiler to integrate the function code into the code of the calling function(s). For more information, see "Inline Function Support Keyword (inline)" on page 1-74.
`asm()`	Places ADSP-21xxx assembly language instructions directly in your C/C++ program. For more information, see "Inline Assembly Language Support Keyword (asm)" on page 1-75.
`dm`	Specifies the location of a static or global variable or qualifies a pointer declaration "*" as referring to Data Memory (DM). For more information, see "Dual Memory Support Keywords (pm dm)" on page 1-90.
`pm`	Specifies the location of a static or global variable or qualifies a pointer declaration "*" as referring to Program Memory (PM). For more information, see "Dual Memory Support Keywords (pm dm)" on page 1-90.
`section("string")`	Specifies the section in which an object or function is placed. The `section` keyword has replaced the `segment` keyword of the previous releases of the compiler software. For more information, see "Placement Support Keyword (section)" on page 1-95.
`bool`, `true`, `false`	A boolean type. For more information, see "Boolean Type Support Keywords (bool, true, false)" on page 1-96.
`restrict` keyword	Specifies restricted pointer features. For more information, see "Pointer Class Support Keyword (restrict)" on page 1-96.

Table 1-8. Operational Extensions

Operation extensions	Description
Variable-length arrays	Support for variable-length arrays lets you use arrays whose length is not known until run time. For more information, see "Variable-Length Array Support" on page 1-97.
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 Initializer Support" on page 1-99.
Indexed initializers	Support for indexed initializers lets you specify elements of an aggregate initializer in arbitrary order. For more information, see "Indexed Initializer Support" on page 1-99.
Aggregate constructor expressions	Support for aggregate assignments lets you create an aggregate array or structure value from component values within an expression. For more information, see "Aggregate Constructor Expression Support" on page 1-101.
`fract` data type (C++ mode)	Support for the fractional data type, fractional and saturated arithmetic. For more information, see "C++ Fractional Type Support" on page 1-146.
Preprocessor-generated warnings	Lets you generate warning messages from the preprocessor. For more information, see "Preprocessor Generated Warnings" on page 1-102.
C++ style comments	Allows for "//" C++ style comments in C programs. For more information, see "C++ Style Comments" on page 1-102.

Inline Function Support Keyword (inline)

The `inline` keyword directs `cc21k` 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 example that follows shows a function definition that uses the `inline` keyword.

```
inline int add_one (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.

In some cases, `cc21k` does not output assembler 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 assembler code.

The compiler only inlines functions, even those declared using the inline keyword, when optimizations are enabled (see "-O[0|1]" on page 1-40).

Inline Assembly Language Support Keyword (asm)

The `cc21k asm()` construct is used to code ADSP-21xxx assembly language instructions within a C or C++ function. The `asm()` construct is useful for expressing assembly language statements that cannot be expressed easily or efficiently with C or C++ constructs.

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

The compiler does not analyze code defined with the `asm()` construct; it passes this code directly to the assembler. The compiler does perform substitutions for operands of the formats `%0` through `%9`; however it passes everything else through to the assembler without reading or analyzing it.

The `asm()` constructs are executable statements, and as such, may not appear before declarations within C/C++ functions.

The `asm()` constructs may also be used at global scope, outside function declarations. Such `asm()` constructs are used to pass declarations and directives directly to the assembler. They are not executable constructs, and may not have any inputs or outputs, or affect any registers.

When optimizing, the compiler sometimes changes the order in which generated functions appear in the output assembly file. However, if global-scope `asm` constructs are placed between two function definitions, the compiler ensures that the function order is retained in the generated assembly file. Consequently, function inlining may be inhibited.

An `asm()` construct without operands takes the form as shown below.

```
asm("nop;");
```

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 (the `-g` switch on page 1-31). These labels are used to generate the debug line information used by the debugger. If the inline assembler inserts conditionally assembled code, an undefined symbol error is likely to occur at link time. For example, the following code could cause undefined symbols if `MACRO` is undefined:

```
asm("#ifdef MACRO");
asm(" // assembly statements");
asm("#endif");
```

If the inline assembler changes the current section and thereby causes the compiler labels to be placed in another section, such as a data section (instead of the default code section), then the debug line information is incorrect for these lines.

Using `asm()` constructs with operands requires some additional syntax described in the following sections.

- "Assembly Construct Template" on page 1-77
- "Assembly Construct Operand Description" on page 1-80
- "Assembly Constructs With Multiple Instructions" on page 1-86
- "Assembly Construct Reordering and Optimization" on page 1-87
- "Assembly Constructs With Input and Output Operands" on page 1-87
- "Assembly Constructs and Flow Control" on page 1-88
- "Guildelines on the Use of asm() Statements" on page 1-89

Assembly Construct Template

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

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

The syntax elements are defined as:

- **template**

  The template is a string containing the assembly instruction(s) with `%number` indicating where the compiler should substitute the operands. Operands are numbered in order of appearance from left to right, starting at 0. Separate multiple instructions with a semicolon, and enclose the entire string within double quotes. For more information on templates containing multiple instructions, see "Assembly Constructs With Multiple Instructions" on page 1-86.

- **constraint**

  The constraint string 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 1-80.

- **output operand**

  The output operands are the names of a C/C++ variables that receives output from a corresponding operand in the assembly instructions.

- **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 overwritten by the assembly instructions. Use lowercase characters to name clobbered registers. Enclose each name within double quotes, and separate each quoted register name with a comma. The input operands are guaranteed not to use any of the clobbered registers, so you can read and write the clobbered registers as often as you like. See Table 1-10 on page 1-85.

asm() Construct Syntax Rules

These rules apply to assembly construct template syntax.

- The template is the only mandatory argument to `asm()`. All other arguments are optional.
- An operand constraint string followed by a C/C++ expression in parentheses describes each operand. For output operands, it must be possible to assign to the expression; that is, the expression must be legal on the left side of an assignment statement.
- A colon separates:
  - The template from the first output operand
  - The last output operand from the first input operand
  - The last input operand from the clobbered registers
- A space must be added 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 SHARC assembly language assignment instruction.

```
{
   int result, x;
   asm (
      "%0= %1;" :
      "=d" (result) :
      "d" (x)
   );
}
```

In the above example, note:

- The template is `"%0= %1;"`. The `%0` is replaced with operand zero `(result)`, the `%1` is replaced with operand one `(x)`.
- The output operand is the C/C++ variable `result`. The letter `d` is the operand constraint for the variable. This constrains the output to a data register `R{0-15}`. The compiler generates code to copy the output from the `R` register to the variable `result`, if necessary. The `=` in `=d` indicates that the operand is an output..

- The input operand is the C/C++ variable `x`. The letter `k` in the operand constraint position for this variable constrains `x` to a data register `R{0-15}`. If `x` is stored in a different kind of register or in memory, the compiler generates code to copy the value into an `R` register before the `asm()` construct uses it.

Assembly Construct Operand Description

The second and third arguments to the `asm()` construct describe the operands in the assembly language template. Several pieces of information must be conveyed for the compiler to know how to assign registers to operands. This information is conveyed with an operand constraint. The compiler needs to know what kind of registers the assembly instructions can operate on, so it can allocate the correct register type.

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

Table 1-9 on page 1-84 describes the correspondence between constraint letters and register classes.

The use of any letter not listed in Table 1-9 results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

To assign registers to the operands, `cc21k` must also be informed 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, such as:

- 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 (see Table 1-9 on page 1-84) describe which registers are modified by an assembly language instruction. The "=" in `=constraint` indicates that the operand is an output; all output operand constraints must use =. Operands that are input-outputs must use "+" (see below).
- The compiler may allocate an output operand in the same register as an unrelated input operand, unless the output operand has the `=&` constraint modifier. This is because `cc21k` 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.

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 is loaded with the value of the C/C++ expression before the `asm()` template is executed. Its C/C++ expression is not modified by the `asm()`, and its value may be a constant or literal.
  Example: `d`

- **= symbol**

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

- **+ symbol**

  The operand is both an input and an output. It must appear as part of the second argument to the `asm()` construct. The allocated register is loaded with the C/C++ expression value, the `asm()` template is executed, and then the allocated register's new value is stored back into the C/C++ expression. Therefore, as with pure outputs, the C/C++ expression must be one that is valid on the left-hand side of an assignment.
  Example: `+d`

- **? symbol**

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

- **& symbol**

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

- **# symbol**

  The operand is an input, but the register's value is clobbered by the `asm()` template execution. The compiler may make no assumptions about the register's final value. An input operand with this constraint will not be allocated the same register as any other input or output operand of the `asm()`. The operand must appear as part of the second argument to the `asm()` construct.
  Example: `#d`

Table 1-9 lists the registers that may be allocated for each register constraint letter. The use of any letter not listed in the “Constraint” column of this table results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter. Table 1-10 on page 1-85 lists the registers that may be named as part of the clobber list

It is also possible to claim registers directly, instead of requesting a register from a certain class using the constraint letters. You can claim the registers directly by simply naming the register in the location where the class letter would be. The register names are the same as those used to specify the clobber list, as shown in Table 1-10.

For example,

```
asm("%0 += %1 * %2;"
   :"=r13"(result)          /* output */
   :"r14"(x),"r15"(y)       /* input  */
);
```

would load `x` into `R14`, load `y` into `R15`, execute the operation, and then store the total from `R13` back into `result`.

Naming the registers in this way allows the `asm()` construct to specify several registers that must be related, such as the DAG registers for a circular buffer. This also allows the use of registers not

covered by the register classes accepted by the `asm()` construct. The clobber string can be any of the registers recognized by the compiler.

Table 1-9. ASM() Operand Constraints

Constraint[1]	Register type	Registers
a	DAG2 B registers	b8 — b15
b	Q2 R registers	r4 — r7
c	Q3 R registers	r8 — r11
d	All R registers	r0 — r15
e	DAG2 L registers	l8 — l15
F	Floating-point registers	F0 — F15
f	Accumulator register	mrf, mrb
h	DAG1 B registers	b0 — b7
j	DAG1 L registers	l0 — l7
k	Q1 R registers	r0 - r3
l	Q4 R registers	r12 - r15
r	All general registers	r0 — r15, i0 — i15, l0 — l15, m0 — m15, b0 — b15, ustat1, ustat2
u	User registers	ustat1, ustat2 (also ustat3, ustat4 on ADSP-2116x, ADSP-2126x and ADSP-2136x processors)
w	DAG1 I registers	I0 — I7
x	DAG1 M registers	M0 — M7
y	DAG2 I registers	I8 — I15
z	DAG2 M registers	M8 — M15
=&constraint	Indicates that the constraint is applied to an output operand that may not overlap an input operand	
=constraint	Indicates that the constraint is applied to an output operand	

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

Constraint[1]	Register type	Registers
&constraint	Indicates the constraint is applied to an input operand that may not be overlapped with an output operand	
?constraint	Indicates the constraint is temporary	
+constraint	Indicates the constraint is both an input and output operand	
#constraint	Indicates that the constraint is an input operand whose value is changed	

1 The use of any letter not listed in Table 1-9 results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

Table 1-10. Register Names for asm() Constructs

Clobber String	Meaning
`"r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7", "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15"`	General data registers
`"i0", "i1", "i2", "i3", "i4", "i5", "i8", "i9", "i10", "i11", "i12", "i13", "i14", "i15"`	Index registers
`"m0", "m1", "m2", "m3", "m4", "m8", "m9", "m10", "m11", "m12"`	Modifier registers
`"b0", "b1", "b2", "b3", "b4", "b7", "b8", "b9", "b10", "b11", "b12", "b13", "b14", "b15",`	Base registers
`"l0", "l1", "l2", "l3", "l4", "l5", "l8", "l9", "l10", "l11", "l12", "l13", "l14", "l15"`	Length registers
`"mrf", "mrb"`	Multiplier result registers
`"acc", "mcc", "scc", "btf"`	Condition registers
`"lcntr"`	Loop counter register
`"PX"`	PX register
`"ustat1", "ustat2"`	User-defined status registers
`"memory"`	Unspecified memory locations

Table 1-10. Register Names for asm() Constructs (Cont'd)

Clobber String	Meaning
The following registers are available on ADSP-2116x/2126x/2136x Processors	
"s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9", "s10", "s11", "s12", "s13", "s14", "s15"	Shadow data registers
"smrf", "smrb"	Shadow multiplier result registers
"sacc", "smcc", "sscc", "sbtf"	Shadow condition registers
"ustat3", "ustat4"	User-defined status registers

Assembly Constructs With Multiple Instructions

There can be many assembly instructions in one template. Your template string may extend over multiple lines. It is not necessary to terminate each line with a backslash (\).

The following listing is an example of multiple instructions in a template.

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

Do not attempt to produce multiple-instruction asm constructs via a sequence of single-instruction asm constructs, as the compiler is not guaranteed to maintain the ordering. For example, the following should be avoided:

```
/* BAD EXAMPLE: Do not use sequences of single-instruction
** asms. Use a single multiple-instruction asm instead.  */

asm("r7=%0;" : : "d" (x) : "r7");
```

```
asm("r6=%0;" : : "d" (y) : "r6");
asm("%0=r6+r7;" : "=d" (result));
```

Assembly Construct Reordering and Optimization

For the purpose of optimization, the compiler assumes that the side effects of an `asm()` construct are limited to changes in the output operands. This does not mean that you cannot use instructions with side effects, but be careful to notify the compiler that you are using them by using the clobber specifiers.

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

```
asm volatile("idle;");
```

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 or C++ statement.

Assembly Constructs With Input and Output Operands

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

one write-only output operand. The connection between the two operands is expressed by constraints in the same location when the instruction executes.

When a register's value is 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("modify(%0,m7);"
   :"=w" (newptr)      // an output, given an I register,
                       // stored into newptr.
   :"0" (oldptr));     // an input, given same reg as %0,
                       // initialized from oldptr
```

Assembly Constructs and Flow Control

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

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

It is also inadvisable to use labels in `asm()` statements, especially when function inlining is enabled. If a function containing such `asm` statements is inlined more than once in a file, there will be multiple definitions of the label, resulting in an assembler error. If possible, use PC-relative jumps in `asm` statements.

Guildelines on the Use of asm() Statements

There are certain operations that are performed more efficiently using other compiler features, and result in source code that is clearer and easier to read.

Accessing System Registers
System registers are accessed most efficiently using the functions in `sysreg.h` instead of using `asm()` statements. For example, the following `asm()` statement:

```
asm("R0 = 0; bit tst MODE1 IRPTEN; if TF r0 = r0 + 1; %0 = r0;"
:  "=d"(test) :  : "r0");
```

can be written as:

```
#include <sysreg.h>
#include <def21060.h>
test = sysreg_bit_tst(sysreg_MODE1, IRPTEN);
```

Accessing Memory-Mapped Registers (MMRs)
MMRs can be accessed using the macros in the `Cdef*.h` files (for example, `Cdef21060.h`) that are supplied with VisualDSP++.

For example, `IOSTAT` can be accessed using `asm()` statements, such as:

```
asm("R0 = 0x1234567; dm(IOSTAT) = R0;" : : : "r0");
```

This can be written more cleanly and efficiently as:

```
#include <Cdef21060.h>
...
*pIOSTAT = 0x1234567;
```

Dual Memory Support Keywords (pm dm)

This section describes cc21k language extension keywords to C and C++ that support the dual-memory space, modified Harvard architecture of the ADSP-21xxx processors. There are two keywords used to designate memory space: `dm` and `pm`. They can be used to specify the location of a static or global variable or to qualify a pointer declaration.

The following rules apply to dual memory support keywords:

- The memory space keyword (`dm` or `pm`) refers to the expression to the right of the keyword.
- 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.
- The compiler always uses Program Memory (PM) as the memory space for functions. Function pointers always point to Program Memory.
- You cannot assign memory spaces to automatic variables. All automatic variables reside on the stack, which is always in Data Memory.
- Literal character strings always reside in Data Memory.

The following listing shows examples of dual memory keyword syntax.

```
int pm buf[100];
/* declares an array buf with 100 elements in Program Memory */
int dm samples[100];
/* declares an array samples with 100 elements in Data Memory */
int points[100];
/* declares an array points with 100 elements in Data Memory */
int pm * pm xy;
```

```
/* declares xy to be a pointer which resides in Program
   Memory and points to a Program Memory integer */
int dm * dm xy;
/* declares xy to be a pointer which resides in Data Memory and
   points to a Data Memory integer */
int *xy;
/* declares xy to be a pointer which resides in Data Memory
   and points to a Data Memory integer */
int pm * dm datp;
/* declares datp to be a pointer which resides in Data Memory
   and points to a Program Memory integer */
int pm * datp;
/* declares datp to be a pointer which resides in Data Memory
   and points to a Program Memory integer */
int dm * pm progd;
/* declares progd to be a pointer which resides in Program
   Memory and points to a Data Memory integer */
int * pm progd;
/* declares progd to be a pointer which resides in Program
   Memory and points to a Data Memory integer */
float pm * dm * pm xp;
/* The first *xp is in Program Memory,
   the following *xp in Data Memory, and xp itself is
   in Program Memory */
```

Memory space specification keywords cannot qualify type names and structure tags, but you can use them in pointer declarations. The following 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    */
```

Memory Keywords and Assignments/Type Conversions

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

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

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

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

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

Memory Keywords and Function Declarations/Pointers

Functions always reside in Program Memory. Pointers to functions always point to Program Memory. The following listing shows some sample function declarations with pointers.

```
/* Dual Memory Support Keyword Function Declaration (With Point-
ers) Syntax Examples */
int * y();          /* function y resides in   */
                    /* pm and returns a        */
                    /* pointer to an integer   */
                    /* which resides in dm     */
int pm * y();       /* function y resides in   */
                    /* pm and returns a        */
                    /* pointer to an integer   */
                    /* which resides in pm     */

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

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

Memory Keywords and Function Arguments

The compiler checks calls to prototyped functions for memory space specifications consistent with the function prototype. The following listing shows sample code that `cc21k` flags as inconsistent use of memory spaces between a function prototype and a call to the function.

```
/* Illegal Dual Memory Support Keywords & Calls To Prototyped
Functions */
extern int foo(int pm*);
```

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

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

foo(&x);    /* call function foo()                             */
            /* using pm pointer (location of x) as the         */
            /* argument. cc21k FLAGS AS AN ERROR; this is an   */
            /* inconsistency between the function's            */
            /* declared memory space argument and function     */
            /* call memory space argument                      */
```

Memory Keywords and Macros

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

To make changes of this type easier, you can use C/C++ preprocessor macros to define common memory spaces that must be coordinated. The following listing shows two code segments that are equivalent after preprocessing. The segment on the right lets you redefine the memory space specifications by redefining the macro `SPACE1` and `SPACE2`.

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

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

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

Placement Support Keyword (section)

The `section` keyword directs the compiler to place an object or function in an assembly `.SECTION` of the compiler's intermediate output file. You name the assembly `.SECTION` directive with the `section()`'s string literal parameter. If you do not specify a `section()` for an object or function declaration, the compiler uses a default section. For information on the default sections, see "Memory Usage" on page 1-177.

Applying `section()` is meaningful only when the data item is something that the compiler can place in the named section. Apply `section()` only to top-level, named objects that have a static duration, are explicitly `static`, or are given as external-object definitions.

The following example shows the declaration of a static variable that is placed in the section called `bingo`.

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

Note that `section` has replaced the `segment` keyword of the Release 4.x compiler. Although the `segment()` keyword is supported by the compiler of the current release, we recommend that you revise the legacy code.

Placement of Compiler-Generated Data

The `-section` switch (on page 1-48) can be used to control the placement of compiler-generated data into non-default sections.

When using certain features of C/C++, the compiler may be required to produce internal data structures. The `section` switch allows you to override the default location where the data would be placed.

For example,

```
cc21k -section vtbl=vtbl_data test.cpp -c++
```

would instruct the compiler to place all the C++ virtual function look-up tables into the section `vtbl_data`, rather than the default `seg_vtbl` section. It is the user's responsibility to ensure that any named sections exist in the `.LDF` file

The following section identifiers are currently supported:

- `vtbl` C++ virtual function lookup tables
- switch tables used in the implementation of switch statements

Boolean Type Support Keywords (bool, true, false)

The `bool`, `true`, and `false` keywords are extensions that support the C++ boolean type. The `bool` keyword is a unique signed integral type, just as the `wchar_t` is a unique unsigned type. There are two built-in constants of this type: `true` and `false`. When converting a numeric or pointer value to `bool`, a zero value becomes `false` and a 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 keyword extensions 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 cannot designate the same object, and therefore, they are not aliases.

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

For example,

```
void fir(short *in, short *c, short *restrict out, int n)
```

The behavior of a program is undefined if it contains an assignment between two restricted pointers, 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 when in C mode (variable-length arrays are not supported for C++). Unlike other automatic arrays, variable-length arrays are declared with a non-constant length. This means that the space is allocated when the array is declared, and 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
tester (int len, char data[len][len])
{
   …
}
```

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

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

As an example, if you were to implement a routine for computation of a product of three matrices, you need to allocate a temporary matrix of the same size as the input matrices. Declaring an 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 the `sizeof()` operator is used to determine the size of the array. For multidimensional arrays, the boundaries are also saved for address computation. After leaving the block, all the space allocated for the array is deallocated. For example, the following program prints 10, not 50.

```
main ()
{
   foo(10);
}
```

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

Non-Constant Initializer Support

The cc21k compiler includes support for the ISO/ANSI standard definition of the C and C++ language and 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

ANSI/ISO standard C/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 cc21k compiler C/C++, by comparison, supports labeling elements for array initializers. This feature lets you specify the array or structure elements in any order by specifying the array indices or structure field names to which they apply. All index values must be constant expressions, even in automatic arrays.

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

```
/* Example 1 Standard & cc21k C/C++ Array Initializer */
/* Standard array initializer */
   int a[6] = { 0, 0, 15, 0, 29, 0 };

/* equivalent cc21k C/C++ array initializer */

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

You can combine this technique of naming elements with standard C/C++ initialization of successive elements. The standard and cc21k 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 & cc21k C/C++ Array Initializer */
/* Standard array initializer */

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

/* equivalent cc21k 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 */
/* cc21k 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 `fieldname:` before the element value. The standard C/C++ and cc21k C/C++ struct initializers in the example below are equivalent.

```
/* Example 4 Standard C & cc21k C/C++ struct Initializer */
/* Standard C struct Initializer */

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

/* Equivalent cc21k C/C++ struct Initializer With
Labeled Elements */

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

Aggregate Constructor Expression Support

Extended initializer support in `cc21k` C/C++ includes support for aggregate constructor expressions, which enable you to assign values to large structure types without requiring each element's value to be individually assigned.

The following example shows an ISO/ANSI standard C `struct` usage followed by equivalent `cc21k` C/C++ code that has been simplified using a constructor expression.

```
/* Standard C struct & cc21k C/C++ Constructor struct */
/* Standard C struct */
   struct foo {int a; char b[2];};
   struct foo make-foo(int x, char *s)
   {
   struct foo temp;
   temp.a = x;
   temp.b[0] = s[0];
   if (s[0] !=  '\0')
      temp.b[1] = s[1];
   else
      temp.b[1] = '\0';
```

```
   return temp;
   }

/* Equivalent cc21k C/C++ constructor struct */
   struct foo make_foo(int x, char *s)
   {
      return((struct foo) {x, {s[0], s[0] ? s[1] : '\0'}});
   }
```

Preprocessor Generated Warnings

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

C++ Style Comments

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

```
a = b

//* highly unusual */ c
```

which a standard C compiler processes as:

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

Compiler Built-In Functions

The compiler supports intrinsic (built-in) functions that enable efficient use of hardware resources. Knowledge of these functions is built into the `cc21k` compiler. Your program uses them via normal function call syntax. The compiler notices the invocation and generates one or more machine instructions, just as it does for normal operators, such as `+` and `*`.

Built-in functions have names which begin with `__builtin_`. Note that identifiers beginning with double underlines (__) are reserved by the C standard, so these names do not conflict with user program identifiers. The header files also define more readable names for the built-in functions without the `__builtin_` prefix. These additional names are disabled if the `-no-builtin` option is used.

This section describes:

- "Access to System Registers" on page 1-103
- "Circular Buffer Built-In Functions" on page 1-107

The `cc21k` compiler provides built-in versions of some of the C library functions as described in "Using Compiler Built-In C Library Functions" on page 3-30.

Access to System Registers

The `sysreg.h` header file defines a set of functions that provide efficient system access to registers, modes, and addresses not normally accessible from C source. These functions are specific to individual architectures.

This section describes the functions that provide access to system registers. These functions are based on underlying hardware capabilities of the ADSP-21xxx processors. The functions are defined in the header file `sysreg.h`. They allow direct read and write access, as well as the testing and modifying of bit sets.

```
int sysreg_read (const int SR_number);
```

`sysreg_read` reads the value of the designated register and returns it.

```
void sysreg_write (const int SR_number, const int new_value);
```

`sysreg_write` stores the specified value in the nominated system register.

```
void sysreg_write_nop (const int SR_number, const int bit_mask);
```

`sysreg_write_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places a '`NOP;`' after the instruction.

```
void sysreg_bit_clr (const int SR_number, const int bit_mask);
```

`sysreg_bit_clr` clears all the bits of the nominated system register that are set in the supplied bit mask.

```
void sysreg_bit_clr_nop (const int SR_number, const int bit_mask);
```

`sysreg_bit_clr_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places '`NOP;`' after the instruction.

```
void sysreg_bit_set (const int SR_number, const int bit_mask);
```

`sysreg_bit_set` sets all the bits of the nominated system register that are also set in the supplied bit mask.

```
void sysreg_bit_set_nop (const int SR_number, const int bit_mask);
```

`sysreg_bit_set_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places '`NOP;`' after the instruction.

```
void sysreg_bit_tgl (const int SR_number, const int bit_mask);
```

`sysreg_bit_tgl` toggles all the bits of the nominated system register that are set in the supplied bit mask.

```
void sysreg_bit_tgl_nop (const int SR_number, const int bit_mask);
```

`sysreg_bit_tgl_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places '`NOP;`' after the instruction.

```
int sysreg_bit_tst (const int SR_number, const int bit_mask);
```

`sysreg_bit_tst` returns a non-zero value if all of the bits that are set in the supplied bit mask are also set in the nominated system register.

```
int sysreg_bit_tst_all (const int SR_number, const int value);
```

`sysreg_bit_tst_all` returns a non-zero value if the contents of the nominated system register are equal to the supplied value.

The register names are defined in `sysreg.h` and must be a compile-time literal. The effect of using the incorrect function for the size of the register or using an undefined register number is undefined.

On all ADSP-21xxx processors, the system registers are:

```
sysreg_IMASK
sysreg_IMASKP
sysreg_ASTAT
sysreg_STKY
sysreg_USTAT1
sysreg_USTAT2
```

In addition, ADSP-2116x, ADSP-2126x and ADSP-2136x processors have the following system registers:

```
sysreg_LIRPTL
sysreg_MMASK
sysreg_ASTATY
sysreg_FLAGS
sysreg_STKYY
sysreg_USTAT3
sysreg_USTAT4
```

Due to hardware characteristics of the SHARC processors, the bit values for the `sysreg_bit_*` interrogation and manipulation functions must be compile-time constants.

For the ADSP-2106x and ADSP-21020 processors, the header files `def21060.h`, `def21061.h`, `def21065l.h` and `def21020.h` provide symbolic names for the individual bits in the system registers.

For ADSP-2116x processors, the header files `def21160.h` and `def21161.h` provide symbolic names for the individual bits in the system registers.

For ADSP-2126x processors, the header files `def21261.h`, `def21262.h`, `def21266.h` and `def21267.h` provide symbolic names for the individual bits in the system registers.

For ADSP-2136x processors, the header files `def21363.h`, `def21364.h` and `def21365.h`, `def21366.h`, `def21367.h`, `def21368.h` and `def21369.h` provide symbolic names for the individual bits in the system registers.

Circular Buffer Built-In Functions

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

Circular Buffer Increment of an Index

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

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

The equivalent operation is:

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

Circular Buffer Increment of a Pointer

The following operation provides a circular buffer increment of an pointer.

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

The equivalent operation is:

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

For more information, see “circindex” on page 3-87 and “circptr” on page 3-89.

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 is within the range `0...(nitems-1)`. When the “-force-circbuf” switch (on page 1-30) is specified, the compiler always treats array references of the form `[ i%n ]` as a circular buffer operation on the array.

Compiler Performance Built-in Functions

These functions provide the compiler with information about the expected behavior of the program. You can use these built-in functions to tell the compiler which parts of the program are most likely to be executed; the compiler can then arrange for the most common cases to be those that execute most efficiently.

```
#include <21060.h>
#include <21065l.h>
#include <21160.h>
#include <21262.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>
int expected_true(int cond);
int expected_false(int cond);
```

For example, consider the code

```
extern int func(int);
int example(int call_the_function, int value)
{
   int r = 0;
   if (call_the_function)
      r = func(value);
   return r;
}
```

If you expect that parameter `call_the_function` to be true in the majority of cases, you can write the function in the following manner:

```
extern int func(int);
int example(int call_the_function, int value)
{
   int r = 0;
   if (expected_true(call_the_function))
                   // indicate most likely true
      r = func(value);
   return r;
}
```

This indicates to the compiler that you expect `call_the_function` to be true in most cases, so the compiler arranges for the default case to be to call function `func()`. If, on the other hand, you were to write the function as:

```
extern int func(int);
int example(int call_the_function, int value)
{
   int r = 0;
   if (expected_false(call_the_function))
                    // indicate most likely false
      r = func(value);
   return r;
}
```

then the compiler arranges for the generated code to default to the opposite case, of not calling function `func()`.

These built-in functions do not change the operation of the generated code, which will still evaluate the boolean expression as normal. Instead, they indicate to the compiler which flow of control is most likely, helping the compiler to ensure that the most commonly-executed path is the one that uses the most efficient instruction sequence.

The `expected_true` and `expected_false` built-in functions only take effect when optimization is enabled in the compiler. They are only supported in conditional expressions.

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 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 these features:

- "Data Alignment Pragmas" on page 1-111
- "Interrupt Handler Pragma" on page 1-115
- "Loop Optimization Pragmas" on page 1-116
- "General Optimization Pragmas" on page 1-119
- "Function Side-Effect Pragmas" on page 1-120
- "Template Instantiation Pragmas" on page 1-127
- "Header File Control Pragmas" on page 1-129
- "Linking Control Pragmas" on page 1-132
- "Diagnostic Control Pragmas" on page 1-135
- "Code Generation Pragmas" on page 1-138

The compiler issues a warning when it encounters an unrecognized pragma directive or pragma operator. The compiler does not expand any preprocessor macros used within any pragma directive or pragma operator. 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 pragmas include `align`, `alignment_region`, `pack` and `pad` pragmas. Alignments specified using these pragmas must be a power of two. The compiler rejects uses of those pragmas that specify alignments that are not powers of 2.

#pragma align *num*

This pragma may be used before variable declarations and field declarations. It applies to the variable or field declaration that immediately follows the pragma.

The pragma's effect is that the next variable or field declaration should be forced to be aligned on a boundary specified by *num*.

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

The pragma also allows the following keywords as allowable alignment specifications:

`_WORD` – Specifies a 32-bit alignment
`_LONG` – Specifies a 64-bit alignment
`_QUAD` – Specifies a 128-bit alignment

The `align` pragma only applies to the immediately-following definition, even if that definition is part of a list. For example,

```
#pragma align 8
int i1, i2, i3;       // pragma only applies to i1
```

#pragma alignment_region (*alignopt*)

Sometimes it is desirable to specify an alignment for a number of consecutive data items rather than individually. This can be done using the `alignment_region` and `alignment_region_end` pragmas:

- `#pragma alignment_region` sets the alignment for all following data symbols up to the corresponding `alignment_region_end` pragma
- `#pragma alignment_region_end` removes the effect of the active alignment region and restores the default alignment rules for data symbols.

The rules concerning the argument are the same as for `#pragma align`. The compiler faults an invalid alignment (such as an alignment that is not a power of two). The compiler warns if the alignment of a data symbol within the control of an `alignment_region` is reduced below its natural alignment (as for `#pragma align`).

Use of the `align` pragma overrides the region alignment specified by the currently active `alignment_region` pragma (if there is one). The currently active `alignment_region` does not affect the alignment of fields.

Example:

```
#pragma align 4

int aa;          /* alignment 4 */
int bb;          /* alignment 1 */

#pragma alignment_region (2)

int cc;          /* alignment 2 */
int dd;          /* alignment 2 */
int ee;          /* alignment 2 */

#pragma align 4
```

```
int ff;          /* alignment 4 */
int gg;          /* alignment 2 */
int hh;          /* alignment 2 */

#pragma alignment_region_end

int ii;          /* alignment 1 */

#pragma alignment_region (3)

long double kk;   /* the compiler faults this, alignment is not
                     a power of two */

#pragma alignment_region_end
```

#pragma pack (*alignopt*)

This pragma may be applied to struct definitions. It applies to all `struct` definitions that follow, until the default alignment is restored by omitting *`alignopt`*; for example, by `#pragma pack()` with empty parentheses.

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

If `alignopt` is specified, it is illegal to invoke `#pragma pad` until the default alignment is restored. The compiler generates an error if the `pad` and `pack` pragmas are used in a manner that conflicts.

#pragma pad (*alignopt*)

This pragma may be applied to `struct` definitions. It applies to all struct definitions that follow, until the default alignment is restored by omitting *`alignopt`*. This pragma is effectively shorthand for placing `#pragma align` before every field within the struct definition. Like the `#pragma pack`, it

reduces the alignment of fields that default to an alignment greater than *alignopt*. However, unlike the #pragma pack, it also increases the alignment of fields that default to an alignment less than *alignopt*.

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

While #pragma pack (*alignopt*) generates a warning if a field alignment is reduced, #pragma pad (*alignopt*) does not. If *alignopt* is specified, it is illegal to invoke #pragma pack, until default alignment is restored.

The following example shows how to use #pragma pad().

```
#pragma pad(4)
struct {
   int i;
   int j;
} s = {1,2};
#pragma pad()
```

Interrupt Handler Pragma

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

The interrupt pragma indicates that the compiler should ensure that all used registers (including scratch registers) are restored at the end of the function. The compiler also ensures that, if an I register is used in the function, the corresponding L register is set to zero, so that circular buffering is not inadvertently invoked.

The `#pragma interrupt` should be used for interrupt handlers which are enabled with the `interruptss()` or `signalss()` family of interrupt functions as these functions ensure that the correct dispatcher is called. For maximum benefit, `#pragma interrupt` should be used for leaf routines only (that is, functions which do not call any other functions). The reason for this is that the interrupt handler ensures that L registers are zeroed for the I registers which are used in the function; if a function call is present, the handler must ensure that all appropriate L registers are set to zero, and this adds considerably to the execution time of the handler.

Loop Optimization Pragmas

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.

The optimizer always attempts to vectorize loops when it is safe to do so. The optimizer exploits the information generated by the interprocedural analysis (see "Interprocedural Analysis" on page 1-67) to increase the cases where it knows it is safe to do so. Consider the following code:

#pragma SIMD_for

This pragma is used with ADSP-2116x, ADSP-2126x and ADSP-2136x processors. It enables the compiler to take advantage of Single-Instruction Multiple-Data (SIMD) operations. See "SIMD Support" on page 1-150 for more details.

#pragma all_aligned

This pragma applies to the subsequent loop. This pragma tells that compiler that all pointer-induction variables in the loop are initially aligned to the maximum permitted alignment of the processor architecture. For ADSP-2106x processors, it is word-aligned; for ADSP-2116x, ADSP-2126x and ADSP-2136x processors, it is double-word aligned.

The variable takes an optional argument (`n`) which can specify that the pointers are aligned after `n` iterations. Therefore, `#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

This pragma is used with ADSP-2116x, ADSP-2126x and ADSP-2136x processors. It ensures the compiler does not generate vectorized SIMD code for the loop on which it is specified.

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

This pragma appears just before the loop it describes. It asserts that the loop iterates at least `min` times, no more than `max` times, and a multiple of `modulo` times. This information enables the optimizer to omit loop guards and to decide whether the loop is worth completely unrolling and whether code needs to be generated for odd iterations. 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++)
```

#pragma no_alias

Use this pragma to tell the compiler that the following loop has no loads or stores that conflict. When the compiler finds memory accesses that potentially refer to the same location through different pointers, known as "aliases", the compiler is restricted in how it may reorder or vectorize the loop, because all the accesses from earlier iterations must be complete before the compiler can arrange for the next iteration to start.

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 `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` is 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.

#pragma vector_for

This pragma notifies the optimizer that it is safe to execute two iterations of the loop in parallel. The `vector_for` pragma does not force the compiler to vectorize the loop. The optimizer checks various properties of the loop and does not vectorize it if it believes it is unsafe or if it cannot deduce that the various properties necessary for the vectorization transformation are valid.

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

```
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.

General Optimization Pragmas

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

- **#pragma optimize_off**
  This pragma turns off the optimizer, if it was enabled, meaning it has the same effect as compiling with no optimization enabled. 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 `cc21k` 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 */
```

Function Side-Effect Pragmas

The function side-effect pragmas (`alloc`, `pure`, `const`, `regs_clobbered`, and `result_alignment`) 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 alloc

This pragma tells the compiler that the function behaves like the library function "malloc", returning a pointer to a newly allocated object. An important property of these functions is that the pointer returned by the function does not point at any other object in the context of the call. In the example,

```
#define N 100

#pragma alloc
int *new_buf(void);
int *vmul(int *a, int *b) {
   int *out = new_buf();
   for (i = 0; i < N; ++i)
      out[i] = a[i] * b[i];
   return out;
}
```

the compiler can reorder the iterations of the loop because the `#pragma alloc` tells it that `a` and `b` cannot overlap `out`.

The GNU attribute `malloc` is also supported with the same meaning.

#pragma const

This pragma is a more restrictive form of the `pure` pragma. The pragma 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 noreturn

This pragma can be placed before a function prototype or definition. Its use tells the compiler that the function to which it applies will never return to its caller. For example, a function such as the standard C function “exit” never returns.

The use of this pragma allows the compiler to treat all code following a call to a function declared with the pragma as unreachable and hence removable.

```
#pragma noreturn
void func() {
   while(1);
}

main() {
   func();
   /* any code here will be removed */
}
```

#pragma pure

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

As this means the function call has the same effect every time it is called, between assignments to global variables, the compiler does not need to 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 regs_clobbered *string*

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

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

```
#pragma regs_clobbered "r4 r8 i4"
void f(void);
```

the compiler knows that only registers `r4`, `r8` and `i4` 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 "r3 m4 r5"
// Function "g" will only clobber r3, m4 and r5
int g(int a) {
   return a+3;
}
```

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.

To obtain optimum results with the pragma, it is best to restrict the clobbered set to be a subset of the default scratch registers. When considering when to apply the `regs_clobbered` pragma, it may be useful to look at the output of the compiler to see how many scratch registers were used. Restricting the volatile set to these registers will produce no impact on the code produced for the function but may free up registers for the caller to allocate across the call site.

The `regs_clobbered` pragma cannot be used in any way with pointers to functions. A function pointer cannot be declared to have a customized clobber set, and it cannot take the address of a function which has a customized clobber set. The compiler raises an error if either of these actions are attempted.

String Syntax

A `regs_clobbered` string consists of a list of registers, register ranges, or register sets that are clobbered (Table 1-11). 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.

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 1-11 shows defined clobbered register 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.

Table 1-11. Clobbered Register Sets

Set	Registers
CCset	ASTAT, ASTATy (ADSP-2116x and ADSP-2126x processors only)
SHADOWset	All S regs, all Shadow MR regs, ASTATy. Always clobbered whether specified or not.
MRset	MRF, MRB; shadow MRF, shadow MRB (ADSP-2116x and ADSP-2126x processors only)
DAG1scratch	Members of DAG1 I, M, B and L-registers that are scratch by default
DAG2scratch	Members of DAG2 I, M, B and L-registers that are scratch by default
DAGscratch	All members of DAG1scratch and DAG2scratch
Dscratch	All D-registers that are scratch by default, ASTAT
ALLscratch	Entire default scratch register set

Unclobberable and Must Clobber Registers

There are certain caveats as to what registers may or must be placed in the clobbered set (Table 1-11). On SHARC processors, certain registers 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 these registers in the clobbered set, a warning is issued and they are removed from the specified clobbered set.

```
I16, I7, B6, B7, L6, L7
```

Registers from these classes,

```
D, I, B, USTAT, LCNTR, PX, MR
```

may be specified in the clobbered set and code is generated to save them as necessary.

The L-registers are required to be set to zero on entry and exit from a function. A user may specify that a function clobbers the L-registers. If it is a compiler-generated function, then it leaves the L-registers at zero at

the end of the function. If it is an assembly function, then it may clobber the L-registers. In that case, the L-registers are re-zeroed after any call to that function. The soft-wired registers `M5,M6,M7` and `M13,M14,M15` are reset in an analogous manner.

The registers `R2` and `I12` are always clobbered. If the user specifies a function definition with the `regs_clobbered` pragma that does not contain these registers, a warning is issued and these registers are added to the clobbered set.

User-Reserved Registers
User-reserved registers, which are indicated via the `-reserve` switch (on page 1-47), are never preserved in the function wrappers whether in the clobbered set or not.

Function Parameters
Function calling conventions are visible to the caller and do not affect the clobbered set that may be used on a function. 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 `R4` and `R8`, respectively. No matter what happens in function `f`, after the call returns, the values of `R4` and `R8` are still set to 2 and 3, respectively.

Function Results
The registers in which a function returns its result must always be clobbered by the callee and retain their new value in the caller. They may appear in the clobbered set of the callee but it makes no difference to the generated code—the return register are not saved and restored. Only the return register used by the particular function return type is special. Return registers used by different return types are treated in the clobbered list in the convention way.

For example,

```
typedef struct { int x, int y } Point;
typedef struct { int x[10] } Big;
int f();  // Result in R0. R1 may be preserved across call
Point g(); // Result in R0 and R1
Big f();  // Result pointer in R0, R1 may be preserved
             across call.
```

#pragma result_alignment (*n*)

This pragma asserts that the pointer or integer returned by the function has a value that is a multiple of *n*.

The pragma is often used in conjunction with the `#pragma alloc` of custom-allocation functions that return pointers that are more strictly aligned than could be deduced from their type.

Template Instantiation Pragmas

The template instantiation pragmas (`instantiate`, `do_not_instantiate` and `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 called instantiation. As templates are a feature of C++, these pragmas are allowed only in `-c++` mode.

Refer to "Compiler C++ Template Support" on page 1-238 for more information on how the compiler handles templates.

These pragmas take the name of an instance as a parameter, as shown in Table 1-12.

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

Table 1-12. Instance Names

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

#pragma instantiate *instance*

This pragma requests the compiler to instantiate *instance* in the current compilation. For example,

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

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

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

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

#pragma do_not_instantiate *instance*

This pragma directs the compiler not to instantiate *instance* in the current compilation. For example,

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

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

#pragma can_instantiate *instance*

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

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

Header File Control Pragmas

The header file control pragmas (`hdrstop`, `no_implicit_inclusion`, `no_pch`, `once`, and `system_header`) help the compiler to handle header files.

#pragma hdrstop

This pragma is used with the `-pch` (precompiled header) switch ("-pch" on page 1-43). The switch tells the compiler to look for a precompiled header (`.pch` file), and, if it cannot find one, to generate a file for use on a later compilation. The `.pch` file contains a snapshot of all the code preceding the header stop point.

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

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

int i;
```

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

For the following example,

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

int i;
```

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

#pragma no_implicit_inclusion

When the `-c++` switch (on page 1-22) is used for each included `.h` file, the compiler attempts to include the corresponding `.c` or `.cpp` file. This is called implicit inclusion.

If `#pragma no_implicit_inclusion` is placed in an `.h` file, the compiler does not implicitly include the corresponding `.c` or `.cpp` file with the `-c++` switch. This behavior only affects the `.h` file with `#pragma no_implicit_inclusion` within it and the corresponding `.c` or `.cpp` files.

For example, if there are the following files

`t.c` containing

```
#include “m.h”
```

and `m.h and m.c` are both empty, then

```
cc21k -c++ t.c -M
```

shows the following dependencies for `t.c`:

```
t.doj: t.c
t.doj: m.h
t.doj: m.C
```

If the following line is added to `m.h`,

```
#pragma no_implicit_inclusion
```

running the compiler as before would not show `m.c` in the dependencies list, such as:

```
t.doj: t.c
t.doj: m.h
```

#pragma no_pch

This pragma overrides the `-pch` (precompiled headers) switch (on page 1-43) for a particular source file. It directs the compiler not to look for a `.pch` file and not to generate one for the specified source file.

#pragma once

This pragma, which should appear at the beginning of a header file, tells the compiler that the header is written in such a way that including it several times has the same effect as including it once. For example,

```
#pragma once
#ifndef FILE_H
#define FILE_H
... contents of header file ...
#endif
```

In this example, the `#pragma once` is actually optional because the compiler recognizes the `#ifndef/#define/#endif` idiom and does not reopen a header that uses it.

#pragma system_header

This pragma identifies an `include` file as the file supplied with VisualDSP++.

The pragma tells the compiler that every function and variable declared in the file (but not in files included in the file) is the variable or function with that name from the VisualDSP++ library. The compiler takes advantage of any special knowledge it has of the behavior of the library.

Linking Control Pragmas

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

#pragma linkage_name *identifier*

This pragma associates the identifier with the next external function declaration. It ensures that the identifier is used as the external reference, instead of following the compiler's usual conventions. For example,

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

#pragma retain_name

This pragma indicates that the function or variable declaration that follows the pragma is not removed even though inter-procedural analysis sees that is not used. Use this pragma before C/C++ function declarations that are only called from assembly routines.

For example,

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

#pragma section/#pragma default_section

The section pragmas provide greater control over the sections in which the compiler places symbols.

The `section(`*`SECTSTRING`* `[, `*`QUALIFIER`*`, ...])` pragma is used to override the target section for any global or static symbol immediately following it. The pragma allows greater control over section qualifiers compared to the `section` keyword.

The `default_section(`*`SECTKIND`* `[, `*`SECTSTRING`* `[, `*`QUALIFIER`*`, ...]])` pragma is used to override the default sections in which the compiler is placing its symbols. The default sections fall into five different categories (listed under *`SECTKIND`*), and this pragma remains in force for a section category until its next use with that particular category. The omission of a section name results in the default section being reset to be the section that was in use at the start of processing.

`SECTKIND` can be one of the following keywords:

`CODE`	Section is used to contain procedures and functions
`ALLDATA`	Section is used to contain any data (normal, read-only and uninitialized)
`DATA`	Section is used to contain “normal data”
`CONSTDATA`	Section is used to contain read-only data
`BSZ`	Section is used to contain uninitialized data

`SECTSTRING` is the double-quoted string containing the section name, exactly as it appears in the assembler file.

`QUALIFIER` can be one of the following keywords:

`PM`	Section is located in program memory
`DM`	Section is located in data memory
`ZERO_INIT`	Section is zero-initialized at program startup
`NO_INIT`	Section is not initialized at program startup
`RUNTIME_INIT`	Section is user-initialized at program startup

DOUBLE32	Section may contain 32-bit but not 64-bit doubles
DOUBLE64	Section may contain 64-bit but not 32-bit doubles
DOUBLEANY	Section may contain either 32-bit or 64-bit doubles

There may be any number of comma-separated section qualifiers within such pragmas, but they must not conflict with one another. Qualifiers must also be consistent across pragmas for identical section names, and omission of qualifiers is not allowed even if at least one such qualifier has appeared in a previous pragma for the same section. If any qualifiers have not been specified for a particular section by the end of the translation unit, the compiler uses default qualifiers appropriate for the target processor. The compiler always tries to honor the section pragma as its highest priority, and the default_section pragma is always the lowest priority.

For example, the following code results in function f being placed in section foo:

```
#pragma default_section(CODE, "bar")
#pragma section("foo")
void f() {}
```

The following code results in x being placed in section zeromem:

```
#pragma default_section(BSZ, "zeromem")
int x;
```

However, the following example does not result in the variable a being placed in section onion because it was declared with the __pm qualifier and therefore is placed in the PM data section:

```
#pragma default_section(DATA, "onion")
__pm int a = 4;
```

#pragma weak_entry

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

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

```
#pragma weak_entry
int w_var = 0;

#pragma weak_entry
void w_func(){}
```

Diagnostic Control Pragmas

The compiler supports `#pragma diag(action: diag [, diag ...])` which allows selective modification of the severity of compiler diagnostic messages.

The directive has three forms:

- modify the severity of specific diagnostics
- modify the behavior of an entire class of diagnostics
- save or restore the current behavior of all diagnostics

Modifying the Severity of Specific Diagnostics

This form of the directive has the following syntax:

```
#pragma diag(ACTION: DIAG [, DIAG ...])
```

The *action:* qualifier can be one of the following keywords:

Keyword	Action
`suppress`	Suppresses all instances of the diagnostic
`remark`	Changes the severity of the diagnostic to a remark.

`warning`	Changes the severity of the diagnostic to a warning.
`error`	Changes the severity of the diagnostic to an error.
`restore`	Restores the severity of the diagnostic to what it was originally at the start of compilation after all command-line options were processed.

The *`diag`* qualifier can be one or more comma-separated compiler diagnostic numbers without the preceding "`cc`" (but can include leading zeros). The choice of error numbers is limited to those that may have their severity overridden (such as those that are displayed with a "`{D}`" in the error message). In addition, those diagnostics that are emitted by the compiler backend (for example, after lexical analysis and parsing) cannot have their severity overridden either. Any attempt to override diagnostics that may not have their severity changed is silently ignored.

Modifying the Behavior of an Entire Class of Diagnostics

This form of the directive has the following syntax:

```
#pragma diag(ACTION)
```

The effects are as follows:

- **#pragma diag*(errors)***

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

- **#pragma diag*(remarks)***

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

- **#pragma diag(*warnings*)**

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

Saving or Restoring the Current Behavior of All Diagnostics

This form has the following syntax:

```
#pragma diag(ACTION)
```

The effects are as follows:

- **#pragma diag(push)**

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

- **#pragma diag(pop)**

  This pragma restores all diagnostic error messages that was previously saved with the most recent `push`.

All `#pragma diag(push)` directives must be matched with the same number of `#pragma diag(pop)` directives in the overall translation unit, but need not be matched within individual source files. Note that the error threshold (set by the `remarks`, `warnings` or `errors` keywords) is also saved and restored with these directives.

The duration of such modifications to diagnostic severity are from the next line following the pragma to either the end of the translation unit, the next `#pragma diag(pop)` directive, or the next overriding `#pragma diag()` directive with the same error number. These pragmas may be used anywhere and are not affected by normal scoping rules.

All command-line overrides to diagnostic severity are processed first and any subsequent `#pragma diag()` directives will take precedence, with the restore action changing the severity back to that at the start of compilation after processing the command-line switch overrides.

Note that the directives to modify specific diagnostics are singular (for example, "`error`"), and the directives to modify classes of diagnostics are plural (for example, "`errors`").

Code Generation Pragmas

The code generation pragmas are:

```
#pragma avoid_anomaly_45 on
#pragma avoid_anomaly_45 off
```

When executing code from external SDRAM on the ADSP-21161 processor, conditional instructions containing a `DAG1` data access may not be performed correctly.

The code generation pragmas allow you to initiate (or avoid) the generation of such instructions on a function-by-function basis. The pragmas should be used before a function definition and remain in effect until another variant of the pragma is seen.

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`

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

Statement Expressions

A statement expression is a compound statement enclosed in parentheses. 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;
}
```

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.

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);
```

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 accepts declarations placed in the middle of code. This allows the declaration of local variables to be placed at the point where they are required. Therefore, the declaration can be combined with initialization of the variable.

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 1-132). 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 `cc21k` compiler are described in the sections of this manual describing the corresponding pragmas (see "Pragmas" on page 1-110).

C++ Fractional Type Support

While in C++ mode, the `cc21k` 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 ADSP-21xxx processors.

Fractional values are declared with the `fract` data type. Ensure that your program incudes 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 .. 1`) with a binary point immediately after the sign bit.

Other value ranges are obtained by scaling or shifting. In addition to the arithmetic, assignment, and shift operations, `fract` provides several type-conversion operations.

For more information about supported fractional arithmetic operators, see "Fractional Arithmetic Operations" on page 1-148. For sample programs demonstrating the use of the `fract` type, see Listing 1-3 on page 1-224.

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

Format of Fractional Literals

Fractional literals use the floating-point representation with an "`r`" suffix to distinguish them from floating-point literals, for example, `0.5r`. The `cc21k` compiler validates fractional literal values at run time to ensure they reside within the valid range of values.

Fractional literals are written with the "`r`" suffix to avoid certain precision loss. Literals without an "`r`" are of the type `double`, and are implicitly converted to `fract` as needed. After the conversion of a 32-bit `double` literal to a `fract` literal, the value of the latter may retain only 25 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 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 `cc21k` compiler:

- Standard arithmetic operations on two `fract` items include addition, subtraction, and multiplication.
- Assignment operations include `+=`, `-=`, and `*=`.
- Shift operations include left and right shifts. A left shift is implemented as a logical shift and a right shift is an arithmetic shift. Shifting left by a negative amount is not recommended.
- Comparison operations are supported between two `fract` items.
- Mixed-mode arithmetic has a preference for fract. For more information about the mixed-mode arithmetic, see "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 cast the result of the multiply operation if the error occurs.

Mixed Mode Operations

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

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

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

Saturated Arithmetic

The `cc21k` compiler supports saturated arithmetic for fractional data in the saturated arithmetic mode.

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

The run-time environment does not change the saturation mode of the processor during initialization. The default mode (typically no saturation) is set by the DSP hardware at reset (consult the hardware reference manual of an appropriate processor for the reset state). The mode can be changed by using `set_saturate_mode()` and `reset_saturate_mode()` functions. Each arithmetic operator has its corresponding variant effected in the saturated mode.

For example, `add_sat`, `sub_sat`, `neg_sat`, and so on.

SIMD Support

The ADSP-2116x, ADSP-2126x and ADSP-2136x processors support Single-Instruction, Multiple-Data (SIMD) operations, which, under certain conditions, double the computational rate over ADSP-2106x processors. It is important to gain some understanding of the processor architecture to take advantage of SIMD mode.

This section contains:

- "Using SIMD Mode with Multichannel Data" on page 1-151
- "Using SIMD Mode with Single-Channel Data" on page 1-152
- "Restrictions to Using SIMD" on page 1-154
- "SIMD_for Pragma Syntax" on page 1-155
- "Compiler Constraints on Using SIMD C/C++" on page 1-157
- "Impact of Anomaly #40 on SIMD" on page 1-158
- "Examples Using SIMD C" on page 1-160
- "Performance When Using SIMD C/C++" on page 1-159

The ADSP-2116x, ADSP-2126x and ADSP-2136x processors include a second processing element that has its own register file and computational units. When the second element is active, the processor operates in the SIMD mode—each computation executes on both processing elements, and each element operates independently on different ("multiple") data.

The SIMD mode is effective for performing exactly the same calculations simultaneously on two parallel sets of data. As a special case — which is what the compiler supports—programs can use the SIMD mechanism to perform a single task (such as summing a vector), by dividing it into two parts and doing both parts in parallel, simultaneously.

Also, there are situations where it is effective to perform two copies of a task simultaneously, such as summing two separate vectors.

SIMD processing is intimately linked with the memory model. In Single-Instruction, Single-Data (SISD) processing (ADSP-2106x compatible mode), the processor fetches single values from memory, performs single arithmetic operations, and stores single values back. In ADSP-2116x SIMD mode, each memory reference fetches a pair of values (from the designated address and the following address), one into each of the two compute blocks. Arithmetic instructions are done in pairs, and paired results are written back.

When compiling for the ADSP-2116x, ADSP-2126x and ADSP-2136x processors, the `cc21k` compiler automatically generates SIMD code wherever possible. However, there are occasions when the `cc21k` compiler does not generate SIMD code because the compiler does not have enough information to be sure that it is safe to use SIMD. If such a situation occurs, the compiler generates a warning, specifying the reason the automatic generation of SIMD code was disabled. This situation occurs most often in functions, where the data to be processed is passed as parameters.

The primary causes of disabling automatic SIMD generation are a lack of alias information or a lack of alignment information. Normally, IPA helps to resolve these issues automatically. However, even with IPA, there may be times when the compiler cannot determine if it is safe to use SIMD code. If SIMD code is appropriate, then the `SIMD_for` pragma can be used to indicate this to the compiler.

Using SIMD Mode with Multichannel Data

When processing multichannel data in SIMD mode, the program essentially runs two copies of the algorithm simultaneously. Multichannel processing could be used for a whole program, for processing two modem channels, or for more local processes, such as calculating the sine of two values simultaneously.

Because there are two copies of the algorithm running, there must be two copies of the data as well. Due to the ADSP-2116x/2126x/2136x SIMD memory architecture, the data must be interleaved in memory. The data

for one channel uses only even locations, while data for the other channel uses the corresponding odd locations. Because the processor implicitly doubles the memory references, arranging data in memory can be as simple as allocating twice as much space for all variables. Correct data arrangement in memory depends on the algorithm.

Such a program could increment loop indices by 2 instead of 1, as each fetch has consumed two words of memory. Data that is common to both could be loaded with a broadcast load or duplicated in memory.

The ADSP-2116x, ADSP-2126x and ADSP-2136x processors can handle conditional execution at a single instruction level in SIMD mode and counted loops. Programs cannot do a conditional branch that depends on the result of a computation in SIMD mode because there would be two different results (one for each channel) and the branch must go one way or the other.

The compiler does not provide extended C/C++ support for SIMD mode and multichannel data.

Using SIMD Mode with Single-Channel Data

When processing single-channel data in SIMD mode, most of the program operates in SISD mode. At key places where the program loops over a collection of contiguous data elements, the program enters SIMD mode to perform computations on both processing elements.

For example, a program adds two vectors element by element. The normal SISD code picks up elements one at a time, evaluating

```
c[j] = a[j] + b[j]
```
for each `j`.

Since each element is processed independently, the operation can as well be done in pairs in SIMD mode:

```
c[j] = a[j] + b[j]
```
in one processing element

```
c[j+1] = a[j+1] + b[j+1]
```
in the other element

Note that the loop index now increments by 2.

This kind of processing is an effective use of the SIMD capability.

SIMD processing can also be used when one of the terms is a scalar. In that case, you should make sure that the same scalar value is loaded into both compute blocks, and the rest of the SIMD processing is the same.

A useful variant of the SIMD loop occurs in a form called a reduction. In such a loop, a vector is reduced to a scalar value by the action of the loop. For example, summing a vector or calculating the dot product of two vectors are areas where a program could use reduction. Again, SIMD processing lets the program process the vector on both processor elements at the same time (half in each).

Note that a reduction loop has to compute a single result. To use SIMD processing, the SISD algorithm would be transformed slightly. Two partial results are accumulated with all the even elements contributing to one result and the odds to the other. At the end of the loop, the program must combine the partial results into a final one. The reduction approach has two effects for which you must account:

- If the data is floating-point type, the results likely differ slightly due to floating-point round-off differences.
- The final combination of the partial results takes a little time that detracts from the SIMD performance gain.

In any of the single channel cases, if the array contains an odd number of elements, an extra step of processing is required after the SIMD region. Note that when the number is not known at compile time, the extra step is conditional on a run-time check.

The compiler provides extended C/C++ support for SIMD mode and single channel data.

Restrictions to Using SIMD

Be aware that there are various implications when a program is transformed to process single-channel data in SIMD mode.

- The data and the access pattern must be arranged for SIMD fetches, which are always at immediately adjoining locations. For example, a program that sums every third element of an array or the first column of a multi-dimensional array would not be a good candidate for SIMD, because the second fetch does not pick up the element for the next iteration.
- The data must always be aligned on double-word boundaries when in SIMD mode. The compiler attempts to align arrays properly in memory, so that operations on whole arrays work. Under certain circumstances, it is possible that some arrays which are placed in named sections may be allocated on an odd word boundary, and may therefore be accessed incorrectly in SIMD mode. These circumstances are usually associated with use of the `RESOLVE` directive in an `.LDF` file or with data elimination by the linker. Therefore, it is suggested that the declaration of any data that is placed in a named section and could be accessed in SIMD mode is preceded by a `#pragma align 2` directive.

You must be careful about situations that force misalignment. This can occur by calling a function with an argument that points to an arbitrary location in an array. For instance, a function `func(A[j])` where `func` expects a pointer or array parameter could have a data alignment problem if the value of `j` is odd. In this case, the parameter denotes a misaligned array, and an attempt to use SIMD processing within `func` fails.

Another SIMD failure situation arises when the reference pattern involves odd locations. A reference to `A[j-1]` fails. Similarly, programs cannot have locations that change between even and odd addresses (for example, `A[j+k]`). The FIR loop nest is an example of the latter.

Some ADSP-2116x, ADSP-2126x and ADSP-2136x architectures only have a 32-bit external bus. Because of a shorter bus length, the effect of SIMD accesses to external memory on such architectures is not the same as SIMD accesses to internal memory. If data is placed in external memory, then the "-no-simd" switch (on page 1-39) or the `no_vectorization` pragma (on page 1-117) should be used to disable SIMD access to this data.

You have to be careful about any interaction or dependency between different iterations of the loop in SIMD. This is important because the SIMD processing changes the order of evaluation. The data for iteration `N+1` is actually fetched from memory before the results of iteration `N` are written back. Some programs get wrong answers if done in SIMD.

Looking at the example,

```
a[j] = a[j-1] + 2;
```

the current iteration uses the results from the previous one; if these results have not yet been written back, the current iteration is incorrect.

SIMD_for Pragma Syntax

The code transformation of a loop to run in SIMD mode involves one command. You indicate which loops are suitable for SIMD execution, and the compiler does the rest. Indicating that a loop should execute in SIMD mode takes the form of a `#pragma` command

```
#pragma SIMD_for
```

which is placed ahead of the loop. As a preprocessing directive, this command must be alone on the line similar to a `#define` or `#include` command.

The compiler responds to this pragma by first checking whether the loop meets the SIMD guidelines. If compliant, the compiler transforms the loop so that the processing is done in SIMD mode. Among other things, the transformation involves changing the loop increment to 2, so that the vector elements are processed in SIMD pairs.

If the loop performs a reduction, partial results are calculated and combined at the end. Also, the compiler takes care of duplicating scalar values used within the loop. The following loop uses `#pragma SIMD_for`.

```
float sum, c, A[N];
…
sum = 0;
#pragma SIMD_for
   for (j=0; j<N; j++) {
       sum += c * A[j];
       }
```

The compiler transforms this loop as:

```
   // declare SIMD temporaries
float t_sum[2], t_c[2];
   // initialize both partial sums
t_sum[0] = t_sum[1] = 0;
   // initialize both parts of scalar constant
t_c[0] = t_c[1] = c;
   // ENTER_SIMD_MODE -- set machine mode
for (j=0; j<N; j+=2) {
   t_sum[0] += t_c[0] * A[j];
   //  --  implicit SIMD processing performs:
   // t_sum[1] += t_c[1] * A[j+1];
   }
   // LEAVE_SIMD_MODE
   // combine partial sums
sum =   t_sum[0] + t_sum[1];
```

Compiler Constraints on Using SIMD C/C++

There are a number of conditions that limit when SIMD operations may occur. The compiler attempts to check these conditions and issues warnings or errors when it detects problems or possible problems.

The compiler usually avoids changing a program when the compiler is not certain that the transformed program produces the same results as the original.

The SIMD transformations are handled a bit differently for two reasons.

- You provide explicit direction to use SIMD. Therefore, the compiler assumes that you are aware of what is needed for SIMD operation and that you share responsibility for correct operation.
- Some of the SIMD constraints are difficult or impossible to verify at compile time. Therefore, the compiler is not fully conservative in checking, because if it were, few, if any loops would be accepted.

The compiler checks the conditions that can be checked. In many cases, the compiler can verify that a loop is unacceptable and rejects it for SIMD processing with a warning or error. Rejection occurs when there is an obvious dependency, obvious alignment problems, or a non-unit stride.

In other cases, the determining values are not available at compile time, and the compiler cannot be sure whether the loop can be safely transformed. In such cases, the compiler issues a warning and proceeds.

For some constraints—primarily the proper alignment of arrays that are parameters (arguments) of the function—the compiler assumes that conditions are acceptable and does not issue a warning.

There are two other restrictions on using `SIMD_for` loops:

- Function calls may not be made from within a `SIMD_for` loop.
- All data types used within a `SIMD_for` loop must have single-word base types. Long doubles, doubles in `double-size-64` mode, and `structs` should not to be used within a `SIMD_for` loop.

Impact of Anomaly #40 on SIMD

The SIMD read from internal memory with a Shadow Write FIFO hit does not always function correctly. This anomaly has been identified in the Shadow Write FIFOs that exist between the internal memory array of the ADSP-21160M processor and core /IOP buses that access the memory. (See *ADSP-21160 SHARC DSP Hardware Reference* for more details on shadow register operation).

If performing SIMD reads which cross Long Word Address boundaries (for example, odd Normal Word addresses or non-Long Word boundary aligned Short Word addresses) and the data for the read is in the Shadow Write FIFO, the read results in revision 0.0 behavior for the read.

To avoid the anomaly #40, SIMD operations must always operate on double-word aligned vectors. To accomplish this type of operation, the compiler

- Allocates all static arrays on an appropriate double-word boundary
- Ensures that arrays on the stack are double-word aligned by ensuring the stack is aligned on an even word boundary.
- Generates SIMD operations when it knows it is operating with double-word-aligned elements. It is able to do this when arrays are defined statically or locally, or the arrays are arguments whose properties can be determined by Inter Procedural Analysis. A further requirement is that the initial index and increment are both explicit.

As a result of these precautions, SISD mode is used when array properties and indexing are not “visible” to the optimizer.

The compiler generates a warning when it fails to automatically generate a SIMD operation due to lack of alignment information. The user can then add the `#pragma SIMD_for` in such cases where they can be sure that alignment requirements are satisfied.

Performance When Using SIMD C/C++

When handling multichannel data in SIMD mode, the ADSP-2116x, ADSP-2126x and ADSP-2136x processors can accomplish roughly twice as much useful work as ADSP-2106x processors. This is diluted slightly if data-dependent conditional blocks must be accommodated. The compiler does not support multichannel data operations in C or C++ code.

When handling single channel data in SIMD mode, C and C++ programs using single-channel SIMD portions usually show some performance improvement but fall short of the double-performance level for a variety of reasons.

In measuring the performance improvement in single channel SIMD, it is useful to isolate the SIMD portion and verify that it is performing as intended. The overall program speedup is often bounded by factors outside of the SIMD portion.

Any parallel-processing situation requires a small amount of overhead to coordinate the parallelism, and the ADSP-2116x, ADSP-2126x and ADSP-2136x processors are no exception. Understanding this factor can help you evaluate where SIMD mode is likely to be most beneficial.

Specific items include:

- **Mode change:** The processor must switch into SIMD mode and back out. Each change takes two cycles.
- **Initialization of scalars:** Non-array values must be duplicated in order to have correct values for both processing elements. This takes a few instructions per value. This is a compiler restriction only—assembly programmers may be able to use the broadcast load facility.
- **Collection of partial results:** For reductions such as vector dot-product, the two partial results must be combined at the end. This typically requires a move and a final add or multiply, another 2 or 3 cycles.

All of these are fairly small items and have little effect provided that the size of the SIMD loop is large.

Note, also, that the size of the inner loop is halved when working in SIMD mode. Consider the following example, a filter with 40 coefficients.

- Inner loop before SIMD: 40 iterations
- Inner loop with SIMD: 20 iterations

Because the ADSP-2106x processors can do a dot-product with a single instruction, the loop represents 20 cycles. If the SIMD overhead is 6 cycles, this operation on the ADSP-2116x, ADSP-2126x and ADSP-2136x processors represents an overhead cost of 30%.

Actually, the true cost is a bit higher. Most loops require a little prologue code to achieve full processor efficiency. When the loop size is halved, the relative impact of the prologue—which remains a constant size—is increased. The prologue can lead to the loss of a few more percentage points off the performance.

Examples Using SIMD C

The following are two examples on how to use SIMD.

- "Using SIMD C (Problem Cases—Data Increments)"
- "Using SIMD C (Problem Cases—Data Alignment)" on page 1-162

Using SIMD C (Problem Cases—Data Increments)

In SIMD mode, an assignment to or from a memory location refers to memory location `[A]` for the `PEx` processing element and memory location `[A+1]` for the `PEy` processing element. The `#pragma SIMD_for` takes advantage of these assignments by taking code containing a stride 1 loop, which addresses contiguous memory locations, and turning it into a stride 2 loop, addressing every second memory location.

In a potentially SIMD compatible loop, it is essential that the stride of a loop through an array is 1, so the compiler uses the correct memory locations. Any other value for the stride results in incorrect behavior.

The following matrix multiplication function demonstrates some stride issues.

This code is NOT a valid use of the `SIMD_for` pragma.

```
float *matmul(void *x_input,
              void *y_input,
              void *output,
              int r,
              int s,
              int t) {
   float *ipx, *ipy, *output_new;
   float tmp = 0;
   int i = 0, j = 0, k = 0;
   ipx = (float *) x_input;
   ipy = (float *) y_input;
   output_new = (float *) output;
   for (i = 0; i < r; i++)
   for (k = 0; k < t; k++) {
      tmp = 0;
      #pragma SIMD_for
        for (j = 0; j < s; j++)
        // The next two lines are wrong in SIMD mode
        tmp += ipx[j + (i * s)] * ipy[k + (j * t)];
        output_new[k + (i * r)] = tmp;
     }
     printf("SIMD\n");
     return output_new;
  }
```

The line in **bold** text above reads from the memory location `ipy[k+(j*t)]`. The loop counter, `j`, is multiplied to calculate the offset into the array. In this example, the stride through the array can not be 1, rather it is `t`.

The SIMD and non-SIMD versions of this code address the following memory locations on each iteration of the innermost loop:

non-SIMD	SIMD
`ipy[k]`	`ipy[k], ipy[k+1]`
`ipy[k + t]`	`ipy[k+(2*t)], ipy[k+(2*t)+1]`
`ipy[k + (2*t)]`	`ipy[k+(4*t)], ipy[k+(4*t)+1]`
`ipy[k + (3*t)]`	`ipy[k+(6*t)], ipy[k+(6*t)+1]`
`ipy[k + (4*t)]`	`ipy[k+(8*t)], ipy[k+(8*t)+1`

Each version addresses different memory locations, giving different results.

SIMD C Loop Counter Rules

- If the loop counter is multiplied within a loop to calculate an offset, do not use SIMD.
- If the inner loop counter is used to subscript a multi-dimensional array, it must only be used as the last subscript. Otherwise, do not use SIMD.

Using SIMD C (Problem Cases—Data Alignment)

To work properly, SIMD calculations must only be used on arrays or other data that are double-word aligned. The compiler and libraries have some responsibility in ensuring that arrays meet this condition. All arrays, unions, and structures are guaranteed to be double-word aligned by the compiler, and functions such as `malloc()` only return double-word aligned memory.

There are some conditions in which you are responsible for determining whether the code and data are compatible with SIMD execution. This section describes some of the pitfalls of which you need to be aware.

Using Two-Dimensional Arrays
The following array,

```
int xyz[9][9];
```

would be double-word aligned by the compiler.

Each sub-array, however, would start at an offset of 9 from the previous array, so `xyz[1]`, `xyz[3]`, and subsequent elements would not be double word aligned. Trying to use these sub-arrays in SIMD mode creates problems. If the function `sum()` contains SIMD code, then the following is incorrect:

```
for (i = 0; i < 10; i++)
   total += sum( xyz[i] );
   // leads to SIMD problems
```

SIMD C/C++ Data Alignment Rule
Two-dimensional arrays containing an odd number of rows or columns may lead to problems.

Adding To Array Offsets
The following is an example in which an offset is calculated using outer and inner loop counters. This code is NOT a valid use of the `SIMD_for` pragma.

```
for (k = 0; k < 20; ++k)
   #pragma SIMD_for
   for (i = 0; i < 20-k; ++i)
      output[i] += (input[i] + input[i+k]);
      // leads to SIMD problems
```

In this case, every second iteration of the outer loop (k=1, k=3, etc.) results in incorrect code as the expression `input[i+k]` is a non-double-word aligned location. Note that this loop could be rewritten as:

```
for (k = 0; k < 20; ++k) {
   if (k % 2)
      for (i = 0; i < 20-k; ++i)
```

```
        output[i] += (input[i] + input[i+k]) ;
else
   #pragma SIMD_for
      for (i = 0; i < 20-k; ++i)
          output[i] += (input[i] + input[i+k]) ;
}
```

This SIMD version is only used when `k=0`, `k=2`, and subsequent even elements. This technique does not offer the full performance benefits of SIMD, but does offer about a 50% improvement.

Preprocessor Features

The compiler includes a preprocessor that lets you use preprocessor commands within your C or C++ source. Table 1-13 lists these commands and provides a brief description of each. The preprocessor automatically runs before the compiler.

Table 1-13. Preprocessor Commands

Command	Description
`#define`	Defines a macro or constant.
`#elif`	Sub-divides an `#if ... #endif` pair.
`#else`	Identifies alternative instructions within an `#if ... #endif` pair.
`#endif`	Ends an `#if ... #endif` pair.
`#error`	Reports an error message.
`#if`	Begins an `#if ... #endif` pair.
`#ifdef`	Begins an `#ifdef ... #endif` pair and tests if macro is defined.
`#ifndef`	Begins an `#ifndef ... #endif` pair and tests if macro is not defined.
`#include`	Includes source code from another file.
`#line`	Outputs specified line number before preprocessing.
`#undef`	Removes macro definition.
`#warning`	Reports a warning message.
`#`	Converts a macro argument into a string constant.
`##`	Concatenates two strings.

Preprocessor commands are also useful for modifying the compilation. Using the `#include` command, you can include header files (`.h`) that contain code and/or data. A macro, which you declare with the `#define` preprocessor command, can specify simple text substitutions or complex

substitutions with parameters. The preprocessor replaces each occurrence of the macro reference found throughout the program with the specified value.

The preprocessor is separate from the compiler and has some features that may not be used within your C or C++ source file. For more information, see the *VisualDSP++ 4.0 Assembler and Preprocessor Manual.*

Predefined Preprocessor Macros

The predefined macros that `cc21k` provides are listed below.

__2106x__

When compiling for the ADSP-21060, ADSP-21061, ADSP-21062, or the ADSP-21065L processors, `cc21k` defines `__ADSP2106x__` as 1.

__2116x__

When compiling for the ADSP-21160 or ADSP-21161 processors, `cc21k` defines `__2116x__` as 1.

__2126x__

When compiling for the ADSP-21261, ADSP-21262, ADSP-21266 or ADSP-21267 processors, `cc21k` defines `__2126x__` as 1.

__2136x__

When compiling for the ADSP-21363, ADSP-21364, ADSP-21365, ADSP-21366, ADSP-21367, ADSP-21368, or ADSP-21369 processors, `cc21k` defines `__2136x__` as 1.

__ADSP21000__

`cc21k` always defines `__ADSP21000__` as 1.

__ADSP21020__

cc21k defines __ADSP21020__ as 1 when you compile with the -21020 command-line switch.

__ADSP21060__

cc21k defines __ADSP21060__ as 1 when you compile with the -21060 command-line switch.

__ADSP21061__

cc21k defines __ADSP21061__ as 1 when you compile with the -21061 command-line switch.

__ADSP21062__

cc21k defines __ADSP21062__ as 1 when you compile with the -21062 command-line switch.

__ADSP21065L__

cc21k defines __ADSP21065L__ as 1 when you compile with the -21065L command-line switch.

__ADSP21160__

cc21k defines __ADSP21160__ as 1 when you compile with the -21160 command-line switch.

__ADSP21161__

cc21k defines __ADSP21161__ as 1 when you compile with the -21161 command-line switch.

__ADSP21261__

cc21k defines __ADSP21261__ as 1 when you compile with the -21261 command-line switch.

__ADSP21262__

cc21k defines __ADSP21262__ as 1 when you compile with the -21262 command-line switch.

__ADSP21266__

cc21k defines __ADSP21266__ as 1 when you compile with the -21266 command-line switch.

__ADSP21267__

cc21k defines __ADSP21267__ as 1 when you compile with the -21267 command-line switch.

__ADSP21363__

cc21k defines __ADSP21363__ as 1 when you compile with the -21363 command-line switch.

__ADSP21364__

cc21k defines __ADSP21364__ as 1 when you compile with the -21364 command-line switch.

__ADSP21365__

cc21k defines __ADSP21365__ as 1 when you compile with the -21365 command-line switch.

__ADSP21366__

`cc21k` defines `__ADSP21366__` as 1 when you compile with the `-21366` command-line switch.

__ADSP21367__

`cc21k` defines `__ADSP21367__` as 1 when you compile with the `-21367` command-line switch.

__ADSP21368__

`cc21k` defines `__ADSP21368__` as 1 when you compile with the `-21368` command-line switch.

__ADSP21369__

`cc21k` defines `__ADSP21369__` as 1 when you compile with the `-21369` command-line switch.

__ANALOG_EXTENSIONS__

`cc21k` defines `__ANALOG_EXTENSIONS__` as 1, and the compiler undefines this macro if you compile with `-pedantic` or `-pedantic-errors`.

__cplusplus

`cc21k` defines `__cplusplus` as 1 when you compile in C++ mode.

__DATE__

The preprocessor expands this macro into the preprocessing date as a string constant. The date string constant takes the form `Mmm dd yyyy`. (ANSI standard).

__DOUBLES_ARE_FLOATS__

`cc21k` defines `__DOUBLES_ARE_FLOATS__` as 1 when the size of the `double` type is the same as the single precision `float` type. When the compiler switch `-double-size-64` is used (on page 1-27), the macro is not defined.

__ECC__

`cc21k` always defines `__ECC__` as 1.

__EDG__

`cc21k` always defines `__EDG__` as 1. This signifies that an Edison Design Group front-end is being used.

__EDG_VERSION__

`cc21k` always defines `__EDG_VERSION__` as an integral value representing the version of the compiler's front-end.

__FILE__

The preprocessor expands this macro into the current input file name as a string constant. The string matches the name of the file specified on the compiler's command-line or in a preprocessor `#include` command. (ANSI standard).

__LINE__

The preprocessor expands this macro into the current input line number as a decimal integer constant. (ANSI standard).

_NO_LONGLONG

`cc21k` always defines `_NO_LONGLONG` as 1.

__NO_BUILTIN

`cc21k` defines `__NO_BUILTIN` as 1 when you compile with the `-no-builtin` command-line switch (on page 1-37).

__SIGNED_CHARS__

`cc21k` defines `__SIGNED_CHARS__` as 1. The macro is defined by default,

__SIMDSHARC__

When compiling for ADSP-2116x, ADSP-2126x and ADSP-2136x processors, `cc21k` defines `__SIMDSHARC__` as 1. The `__SIMDSHARC__` define is used to identify processors that are capable of executing SIMD code.

__STDC__

`cc21k` always defines `__STDC__` as 1.

__STDC_VERSION__

`cc21k` always defines `__STD_VERSION__` as 199409L.

__TIME__

The preprocessor expands this macro into the preprocessing time as a string constant. The date string constant takes the form `hh:mm:ss` (ANSI standard).

__VERSION__

The preprocessor expands this macro into a string constant containing the current compiler version.

__VERSIONNUM__

The preprocessor defines __VERSIONNUM__ as a numeric variant of __VERSION__ constructed from the version number of the compiler. Eight bits are used for each component in the version number and the most significant byte of the value represents the most significant version component. As an example, a compiler with version 7.1.0.0 defines __VERSIONNUM__ as 0x07000100 and 7.1.1.10 would define __VERSIONNUM__ to be 0x0701010A.

__WORKAROUNDS_ENABLED

cc21k defines this macro to be 1 if any hardware workarounds are implemented by the compiler. This macro is set if the -si-revision option (on page 1-49) has a value other than "none" or if any specific workaround is selected by means of the -workaround compiler switch (on page 1-56).

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 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
```

C/C++ Run-Time Model and Environment

This section describes the conventions that you must follow as you write assembly code that can be linked with C or C++ code. The description of how C or C++ constructs appear in assembly language are also useful for low-level program analysis and debugging

This section provides a full description of the ADSP-21xxx run-time model, including the layout of the stack, data access, and call/entry sequence.

This section describes:

- "C/C++ Run-Time Environment"
- "Support for argv/argc" on page 1-185
- "Using Multiple Heaps" on page 1-185
- "Compiler Registers" on page 1-192

This model applies to the compiler-generated code. Assembly programmers are encouraged to maintain stack conventions.

C/C++ Run-Time Environment

The C/C++ run-time environment is a set of conventions that C and C++ programs follow to run on ADSP-21xxx processors. Assembly routines that you link to C or C++ routines must follow these conventions.

Figure 1-3 shows an overview of the run-time environment issues that you must consider as you write assembly routines that link with C/C++ routines. These issues include:

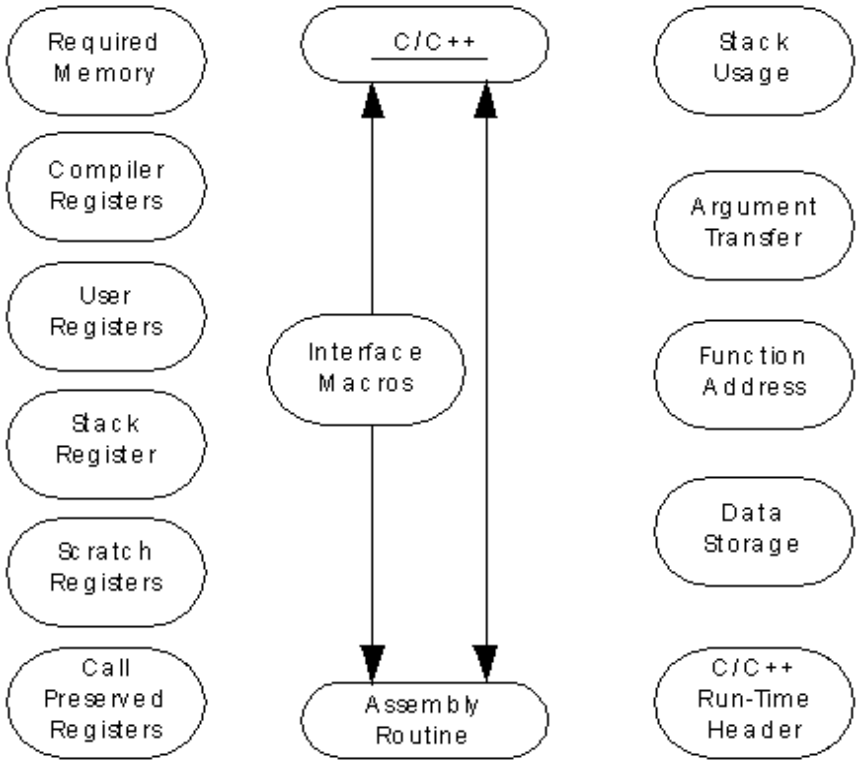


Figure 1-3. Assembly Language Interfacing Overview

- Register usage conventions (see the following sections)

  "Compiler Registers" on page 1-192

  "Miscellaneous Information About Registers" on page 1-193

  "Call Preserved Registers" on page 1-194

  "Scratch Registers" on page 1-195

  "Stack Registers" on page 1-196

- Memory usage conventions (see the following sections)

  "Memory Usage" on page 1-177

  "Measuring the Performance of C Compiler" on page 1-183

  "Using Data Storage Formats" on page 1-205

- Program control conventions (see the following sections)

  "Managing the Stack" on page 1-197

  "Transferring Function Arguments and Return Value" on page 1-202

  "Using Data Storage Formats" on page 1-205

Memory Usage

The `cc21k` C/C++ run-time environment requires that a specific set of memory section names be used for placing code in memory. In assembly language files, these names are used as labels for the `.SECTION` directive. In the Linker Description File (`.LDF`), these names are used as labels for the output section names within the `SECTIONS{}` command.

For information on syntax for the Linker Description File and other information on the linker, see the *VisualDSP++ 4.0 Linker and Utilities Manual*. Table 1-14 lists the memory section and output section names.

Because the compiler and linker must know the processor type to create code for the correct memory model, you must specify the processor for which you are developing. If you are using the VisualDSP++ IDDE, you specify the processor in the Project Options dialog box. If you are running the compiler from the command line, you specify the processor with a compiler switch. For more information on processor selection switches, see "C/C++ Compiler Common Switch Descriptions" on page 1-23.

Table 1-14. Memory .SECTION and SECTION{} Names

Names	Usage Description
seg_pmco	This section must be in Program Memory, holds code, and is required by some functions in the C/C++ run-time library. For more information, see "Program Memory Code Storage"on page 1-178.
seg_dmda	This section must be in Data Memory, is the default location for global and static variables and string literals, and is required by some functions in the C/C++ run-time library. For more information, see "Data Memory Data Storage"on page 1-179.
seg_pmda	This section must be in PM, holds PM data variables, and is required by some functions in the C/C++ run-time library. For more information, see "Program Memory Data Storage" on page 1-179.
seg_stak	This section must be in DM, holds the run-time stack, and is required by the C/C++ run-time environment. For more information, see "Run-Time Stack Storage" on page 1-179.
seg_heap	This section must be in DM, holds the default run-time heap, and is required by the C/C++ run-time environment. For more information, see "Run-Time Heap Storage" on page 1-180.
seg_init	This section must be in PM, holds system initialization data, and is required for system initialization. For more information, see "Initialization Data Storage" on page 1-180.
seg_rth	This section must be in the interrupt table area of PM, holds system initialization code and interrupt service routines, and is required for system initialization. For more information, see "Run-Time Header Storage" on page 1-182.
seg_int_code	This section must always be located in internal memory. It contains library code that modifies the interrupt latch registers (IMASKP and IRPTL). A hardware anomaly on a number of SHARC processors means that it is unsafe for code located in external memory to modify these registers. This section is used to locate the affected library code in internal memory without restricting the location of the rest of the library code.

Program Memory Code Storage

The Program Memory code section, seg_pmco, is where the compiler puts all the program instructions that it generates when you compile your program. When linking, use your .LDF file to map this section to PM space.

Data Memory Data Storage

The Data Memory data section, `seg_dmda`, is where the compiler puts global and static data. When linking, use your `.LDF` file to map this section to DM space

By default, the compiler stores static variables in the Data Memory data section. The compiler's `dm` and `pm` keywords (memory type qualifiers) let you override this default. If a memory type qualifier is not specified, the compiler places static and global variables in Data Memory. For more information on type qualifiers, see "Dual Memory Support Keywords (pm dm)" on page 1-90. The following example allocates an array of 10 integers in the DM data section.

```
static int data [10];
```

Program Memory Data Storage

The Program Memory data section, `seg_pmda`, is where the compiler puts global and static data in Program Memory. When linking, use your `.LDF` file to map this section to PM space.

By default, the compiler stores static variables in the Data Memory data section. The compiler's `pm` keyword (memory type qualifier) lets you override this default and place variables in the Program Memory data section. If a memory type qualifier is not specified, the compiler places static and global variables in Data Memory. For more information on type qualifiers, see "Dual Memory Support Keywords (pm dm)" on page 1-90. The following example allocates an array of 10 integers in the PM data section.

```
static int pm coeffs[10];
```

Run-Time Stack Storage

The run-time stack section, `seg_stak`, is where the compiler puts the run-time stack in Data Memory. When linking, use your `.LDF` file to map this section to DM space. Because the run-time environment cannot function without this section, you must define it, and the section must be in DM space. A typical size for the run-time stack is 4K 32-bit words of Data Memory.

The run-time stack is a 32-bit wide structure, growing from high memory to low memory. The compiler uses the run-time stack as the storage area for local variables and return addresses.

During a function call, the calling function pushes the return address onto the stack (see "Managing the Stack" on page 1-197).

Run-Time Heap Storage
The run-time heap section, `seg_heap`, is where the compiler puts the run-time heap in Data Memory. When linking, use your `.LDF` file to map the `seg_heap` section to Data Memory space. A typical size for the run-time heap is 60K 32-bit words of Data Memory.

To dynamically allocate and deallocate memory at run-time, the C or C++ run-time library includes five functions: `malloc`, `calloc`, `realloc` and `free`. These functions allocate memory from the `seg_heap` section of memory by default.

The run-time library also provides support for multiple heaps, which allow dynamically allocated memory to be located in different blocks. See "Using Multiple Heaps" on page 1-185 for more information on the use of multiple heaps.

The Linker Description File requires the `seg_heap` declaration for every DSP project whether a program dynamically allocates memory at run-time or not.

Initialization Data Storage
The initialization section, `seg_init`, is where the compiler puts the initialization data in Program Memory. When linking, use your Linker Description File to map this section to Program Memory space.

The initialization section may be processed by two different utility programs: `mem21k` or `elfloader`.

- If you are producing boot-loadable executable file for your processor system, you should use the `elfloader` utility to process your executable. The `elfloader` utility processes your executable file, producing an ADSP-2106x boot-loadable file which you can use to boot a target hardware system and initialize its memory.

  The boot loader, `elfloader`, operates on the executable file produced by the linker. When you run `elfloader` as part of the compilation process (using the `-no-mem` switch), the linker (by default) creates a `*.dxe` file for processing with `elfloader`.

  When preparing files for the `elfloader` loader, the system configuration file's `seg_init` section needs only 16 slots/locations of space.

- If producing an executable file that is not going to be boot-loaded into the processor, you may use the `mem21k` utility to process your executable. The `mem21k` utility processes your executable file, producing an optimized executable file in which all RAM memory initialization is stored in the `seg_init` PM ROM section. This optimization has the advantage of initializing all RAM to its proper value before the call to `main()` and reducing the size of an executable file by combining contiguous, identical initializations into a single block.

  The memory initializer, `mem21k`, operates on the executable file produced by the linker. When running `mem21k` as part of the compilation process, the linker (by default) creates a `*.dxe` file for processing with `mem21k`.

  The `mem21k` utility processes all the `PROGBITS` and `ZERO_INIT` sections except the initialization section (`seg_init`), the run-time header section (`seg_rth`), and the code section (`seg_pmco`). These sections contain the initialization routines and data.

The C run-time header reads the `seg_init` section, generated by `mem21k`, to determine which memory locations should be initialized to what values. This process occurs during the `__lib_setup_processor` routine that is called from the run-time header.

Run-Time Header Storage

The run-time header section, `seg_rth`, is where the compiler puts the system initialization code and interrupt table in Program Memory. When linking, use your `.LDF` file to map this section to the interrupt vector table area of Program Memory space.

If a run-time header file is not specified, the compiler uses a default run-time header from the appropriate `...lib` directory, such as:

TARGET	HEADER FILE
21020	21k\lib\020_hdr.doj
21060	21k\lib\060_hdr.doj
21061	21k\lib\061_hdr.doj
21062	21k\lib\060_hdr.doj
21065L	21k\lib\065L_hdr.doj
21160	211xx\lib\160_hdr.doj
21161	211xx\lib\161_hdr.doj
21261	212xx\lib\261_hdr.doj
21262	212xx\lib\262_hdr.doj
21266	212xx\lib\266_hdr.doj
21267	212xx\lib\267_hdr.doj
21363	213xx\lib\363_hdr.doj
21364	213xx\lib\364_hdr.doj
21365	213xx\lib\365_hdr.doj

TARGET	HEADER FILE
21366	213xx\lib\366_hdr.doj
21367	213xx\lib\367_hdr.doj
21368	213xx\lib\368_hdr.doj
21369	213xx\lib\369_hdr.doj

Note that if the compiler finds a copy of `xxx_hdr.obj` in the current directory, the compiler uses this copy instead of the file from the default directory.

The source files for many run-time header files (including `060_hdr.asm` and `160_hdr.asm`) come with the development tools package. Keep the following points in mind if you prefer to write your own interrupt handlers in C or C++:

- Note that the library functions `signal`, `raise`, `interrupt`, and their variants are based on the run-time header used.
- Note that on the ADSP-21020 processor only, each interrupt is allocated eight words; on all other SHARC processors, each interrupt is allocated four words.

Measuring the Performance of C Compiler

Benchmarking is done to measure the performance of a C compiler, or to understand how many processor clock cycles a specific section of code usually takes. Once the number of clock cycles is known, the amount of time that function takes to execute can be quickly calculated using the instruction rate of that processor.

SHARC processors have a set of registers called `EMUCLK` and `EMUCLK2` which make up a 64-bit counter. This counter is unconditionally incremented during every instruction cycle on the processor and is not affected by cache-misses, wait-states, etc. Every time `EMUCLK` wraps back to zero,

EMUCLK2 is incremented by one. These registers, while not documented, are great for benchmarking purposes and can be accessed like any universal register (for example, r0 = EMUCLK;).

Unfortunately, these variables are not directly accessible from C as they are not memory-mapped. Therefore, two macros are provides to retrieve the EMUCLK values and compute cycle counts.

The CYCLE_COUNT_START and CYCLE_COUNT_STOP macros can be copied and pasted into any C file and used freely. For example,

```
#define CYCLE_COUNT_START( cntr ) asm("r0 = emuclk; %0 = r0;": \
                                      "=k" (cntr):"d" (cntr): \
                                      "r0")
#define CYCLE_COUNT_STOP( cntr ) asm("r0 = emuclk; r1 = %1; r2 = 4; \
                           r0 = r0 - r2; r0 = r0 - r1; %0 = r0;" : \
                           "=k" (cntr) :                          \
                           "d" (cntr) : "r0", "r1")
```

The first macro, CYCLE_COUNT_START, copies the contents of EMUCLK into an integer variable.

The second macro, CYCLE_COUNT_STOP, subtracts the stored value of EMUCLK from the current value. It then subtracts a value of 4 representing the overhead incurred in these two functions. The final value is then stored back into the integer variable.

These macros do not take into account the potential wrapping of the EMUCLK register back to zero and the subsequent increment of the EMUCLK2 register in order to save space and execution time. If benchmarking is to be done on a free-running system, the EMUCLK register wraps once every 71 seconds on a 60MHz processors. However, since most benchmarking is typically done in a simulation environment or in a controlled emulation environment, a EMUCLK wrap is typically never encountered as these registers are reset with the processor.

Support for argv/argc

By default, the facility to specify arguments that get passed to your `main()` (`argv/argc`) at run-time is enabled. However, to correctly set up `argc` and `argv` requires additional configuration by the user. You need to modify your application in the following ways:

1. Define your command-line arguments in C by defining a variable called "`__argv_string`". When linked, your new definition overrides the default zero definition otherwise found in the C run-time library. For example,

```
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 do this for you if macro `IDDE_ARGS` is defined at link time.

Using Multiple Heaps

The SHARC 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, which can be located in any of the SHARC processor memory blocks. These additional heaps can be accessed either by the standard `calloc`, `free`, `malloc`, and `realloc` functions, or via the Analog Devices extensions `heap_calloc`, `heap_free`, `heap_malloc`, and `heap_realloc`.

The primary use of alternate heaps is to allow dynamic memory allocation from more than one memory block. The ADSP-21xxx architecture allows two data accesses per cycle (in addition to a code access) if the memory locations are in different blocks.

Declaring a Heap

Each heap must be declared with a `.VAR` directive in the `seg_init.asm` file and the `.LDF` file must declare memory and section placement for the heaps. The default `seg_init.asm` file declares one heap, `seg_heap`. The following customized `seg_init.asm` file shows how to declare two heaps: `seg_heap` and `seg_heaq`.

To use a custom `seg_init.asm`, assemble it and use it to replace the default `seg_init.doj` that is a member of the `libc.dlb` archive. For information on how to modify an archive file, see the *VisualDSP++ 4.0 Linker and Utilities Manual*.

For example,

```
.segment/pm seg_init;

.extern ldf_heap_space;   /* The base of a primary DM heap
"seg_heap" */
.extern ldf_heap_length;  /* Length of heap "seg_heap" */
.extern ldf_heaq_space;   /* Base of a primary DM heap "seg_heaq" */
.extern ldf_heaq_length;  /* Length of heap "seg_heaq" */

/* The first two 48-bit words represent the heap name and the
heap location */
/*   FFFFFFFF DM location   00000001 PM location */
/* The heap name must be exactly 8 characters long */
/* The next 3 words set the heap's initialization value, size
and length. */
/* The size and length are set with macros whose values are
calculated according the information in the project's .ldf file. */

.global ___lib_heap_space;
```

```
.var    ___lib_heap_space[5] =
        0x7365675F6865, /* 'seg_he' */
        0x6170FFFFFFFF, /* 'ap'     */
        0,
        ldf_heap_space,
        ldf_heap_length;
.___lib_heap_space.end:

/* Add more heap descriptions here */
.global ___lib_heaq_space;
.var    ___lib_heaq_space[5] =
        0x7365675F6865, /* 'seg_he' */
        0x6171FFFFFFFF, /* 'aq'     */
        0,
        ldf_heaq_space,
        ldf_heaq_length;
.___lib_heaq_space.end:

.var    ___lib_end_of_heap_descriptions = 0;
        /* Zero for end of list */

.___lib_end_of_heap_descriptions.end:

.endseg;
```

As noted above, 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 `seg_init.asm` file). The following `.LDF` file supports the customized `seg_init.asm` file from the previous example.

```
ARCHITECTURE(ADSP-21062)
SEARCH_DIR( $ADI_DSP\21k\lib )
$LIBRARIES = lib060.dlb, libc.dlb ;
$OBJECTS = $COMMAND_LINE_OBJECTS, adi_dsp\21k\lib\060_hdr.obj, seg_init.doj;

MEMORY {
   seg_rth {TYPE(PM RAM) START(0x20000) END(0x20fff) WIDTH(48)}
   seg_init{TYPE(PM RAM) START(0x21000) END(0x2100f) WIDTH(48)}
   seg_pmco{TYPE(PM RAM) START(0x21010) END(0x24fff) WIDTH(48)}
```

```
    seg_pmda{TYPE(DM RAM) START(0x28000) END(0x28fff) WIDTH(32)}
    seg_dmda{TYPE(DM RAM) START(0x29000) END(0x29fff) WIDTH(32)}
    seg_stak{TYPE(DM RAM) START(0x2e000) END(0x2ffff) WIDTH(32)}
    /* memory declarations for default heap */
    seg_heap{TYPE(DM RAM) START(0x2a000) END(0x2bfff) WIDTH(32)}
    /* memory declarations for custom heap */
    seg_heaq{TYPE(DM RAM) START(0x2c000) END(0x2dfff) WIDTH(32)}
    } // End MEMORY

PROCESSOR p0 {
    LINK_AGAINST( $COMMAND_LINE_LINK_AGAINST)
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

    SECTIONS {
     .seg_rth {
       INPUT_SECTIONS( $OBJECTS(seg_rth) $LIBRARIES(seg_rth))
     } > seg_rth
     .seg_init {
       INPUT_SECTIONS( $OBJECTS(seg_init) $LIBRARIES(seg_init))
     } > seg_init
     .seg_pmco {
       INPUT_SECTIONS( $OBJECTS(seg_pmco) $LIBRARIES(seg_pmco))
     } > seg_pmco
     .seg_pmda {
       INPUT_SECTIONS( $OBJECTS(seg_pmda) $LIBRARIES(seg_pmda))
     } > seg_pmda
     .seg_dmda {
       INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
     } > seg_dmda
     .stackseg {
       ldf_stack_space = .;
       ldf_stack_length = 0x2000;
     } > seg_stak

     /* section placement for default heap */
     .heap {
       ldf_heap_space = .;
       ldf_heap_end = ldf_heap_space + 0x2000;
       ldf_heap_length = ldf_heap_end - ldf_heap_space;
     } > seg_heap
```

```
    /* section placement for additional custom heap */
    .heap {
      ldf_heaq_space = .;
      ldf_heaq_end = ldf_heaq_space + 0x2000;
      ldf_heaq_length = ldf_heaq_end - ldf_heaq_space;
    } > seg_heaq
  } // End SECTIONS
} // End PO
```

Heap Identifiers

All heaps have two identifiers:

- Primary heap ID is the index of the descriptor for that heap in the heap descriptor table (in `seg_init.asm`). The primary heap ID of the default heap is always 0, and the primary IDs of user-defined heaps are set to 1, 2, 3, and so on.
- Each heap also has a unique 8-letter name associated with it. The heap ID can be obtained by calling the function `heap_lookup_name` with this name as its parameter. The name must be exactly eight characters long.

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 a current heap, which initially is the default heap. The current heap can be changed any number of times at runtime by calling the function `set_alloc_type` with the new heap name as a parameter, or 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 also allocates a new object from the current heap.

Previously allocated objects can be deallocated with `free` or `realloc`, or resized by `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 is always in the same heap as the original object.

Multithreaded programs (using VDK) cannot use `set_alloc_type` or `heap_switch` to change the current heap from the default. Such programs can access alternate heaps through the alternate interface described in the next section.

Using the Alternate Heap Interface

The C run-time library provides the alternate heap interface functions `heap_calloc`, `heap_free`, `heap_malloc`, and `heap_realloc`. These routines work 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 does not 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 won't be freed or reallocated.

The actual entry point names for the alternate heap interface routines have an initial underscore; they are `_heap_calloc`, `_heap_free`, `_heap_lookup`, `_heap_malloc`, `_heap_realloc` and`_heap_switch`. The `stdlib.h` standard header file defines macro equivalents without the leading underscores.

Example C Programs

The C programs below show how to allocate and initialize heaps.

Standard Heap Interface

```
// Example program using the standard heap interface
// Assumes that the user has created an additional heap, "seg_heaq",
// which is located in PM memory

#include <stdlib.h>
#include <stdio.h>

void func(int * a, int pm * b);

main()
{
    int * x;
    int pm * y;
    int loop;

    x = malloc(1000);            // get 1K words of DM heap space
    set_alloc_type("seg_heaq"); // Set the current heap to "seg_heaq"
    y = (int pm *)malloc(1000); // get 1K words of PM heap space
    set_alloc_type("seg_heap"); // Reset the current heap to
                                // "seg_heap" in case it is referred
                                // to elsewhere
    for (loop = 0; loop < 1000; loop++)
        x[loop] = y[loop] = loop;
    func(x, y);                // Do something with x and y
}
```

Alternate Heap Interface

```
// Example function using the alternate heap interface
// Assumes that the user has created an additional heap, "seg_heaq",
// which is located in PM memory
#include <stdlib.h>

void func(int * a, int pm * b);
```

```
main()

{
    int * x;
    int pm * y;
    int loop, pm_heapID;
    pm_heapID = heap_lookup_name("seg_heaq");
    x = heap_malloc(0, 1000);  // get 1K words of DM heap space
    y = (int pm *)heap_malloc(pm_heapID, 1000);
                               // get 1K words of PM heap space
    for (loop = 0; loop < 1000; loop++)
        x[loop] = y[loop] = loop;
    func(x, y);                // Do something with x and y
}
```

Compiler Registers

The `cc21k` C/C++ run-time environment reserves a set of registers for its own use. Table 1-15 lists these registers and the values the C/C++ run-time environment expects to be in them. Do not modify these registers, except as noted in the table.

Table 1-15. Compiler Registers

Register	Value	Modification Rules
m5, m13	0	Do not modify
m6, m14,	1	Do not modify
m7, m15	-1	Do not modify
b6, b7	stack base	Do not modify
l6, l7	stack length	Do not modify
l0, l1, l2, l3, l4, l5, l8, l9, l10, l11, l12, l13, l14, l15	0	Modify for temporary use, restore when done
MMASK (ADSP-2116x/26x/36x processors only)	0xE03003	Do not modify if you are using the interrupt dispatchers supplied with VisualDSP++.

Miscellaneous Information About Registers

The following is some miscellaneous information that you might find helpful in understanding register functionality:

- All of the L registers, except `L6` and `L7`, are required to be zero at any call/return point.
- When you either make a function call or return to your caller and have modified any of the L registers, you must reset them to zero.
- Interrupt routines must save and set to zero the L register before using its corresponding I register for any post-modify instruction.
- The `MMASK` register ensures that `MODE1` is set to the correct value before the interrupt dispatcher code is executed. It ensures that the following bits are cleared: `BR0`, `BR8`, `IRPTEN`, `ALUSAT`, `PEYEN`, `BDCST1`, `BDCST9`.

User Registers

The `-reserve` command-line switch lets you reserve registers for your inline assembly code or assembly language routines. If reserving an L register, you must reserve the corresponding I register; reserving an L register without reserving the corresponding I register can result in execution problems.

You must reserve the same list of registers in all linked files; the whole project must use the same `-reserve` option. Table 1-16 lists these registers. Note that the C run-time library does not use these registers.

When you reserve a register, you are asking the compiler to avoid using the register. If the compiler requires a register you have reserved, the compiler ignores your reservation request. Reserving registers can negatively influence the efficiency of compiled C or C++ code; use this option infrequently.

Table 1-16. User Registers

Register	Value	Modification Rule
i0, b0, l0, m0, i1, b1, l1, m1, i8, b8, l8, m8, i9, b9, l9, m9, mrb, ustat1, ustat2, ustat3, ustat4	user defined	If not reserved, modify for temporary use, restore when done If reserved, usage is not limited

Call Preserved Registers

The `cc21k` C/C++ run-time environment specifies a set of registers whose contents must be saved and restored. Your assembly function must save these registers during the function's prologue and restore the registers as part of the function's epilogue. These registers must be saved and restored if they are modified within the assembly function; if a function does not change a particular register, it does not need to save and restore the register.

Table 1-17 lists these registers.

Table 1-17. Call Preserved Registers[1]

b0	b1	b2	b3	b5	b8
b9	b10	b11	b14	b15	
i0	i1	i2	i3	i5	i8
i9	i10	i11	i14	i15	mode1
mode2	mrb	mrf	m0	m1	m2
m3	m8	m9	m10	m11	r3
r5	r6	r7	r9	r10	r11
r13	r14	r15			

1 If you use a call preserved I register in an assembler routine called from an assembler routine, you must save and zero (clear) the corresponding L register as part of the function prologue. Then, restore the L register as part of the function epilogue.

Many functions in the C/C++ run-time library expect the processor to be in a specific mode and may not operate correctly if the processor is in a different mode. If you need to change processor modes, save the old values in the `mode1` and `mode2` registers and restore these registers before calling or returning to calling functions.

The C/C++ run-time environment:

- Uses default bit order for `DAG` operations (no bit reversal)
- Uses the primary register set (not background set)
- Uses `.PRECISION=32` (32-bit floating-point) and `.ROUND_NEAREST` (round-to-nearest value)
- Disables ALU saturation (`mode1` register, `ALUSAT` bit = 0)
- Uses default `FIX` instruction rounding to nearest (`MODE1` register, `TRUNCATE`=0)
- Enables circular buffering on ADSP-2116x, ADSP-2126x and ADSP-2136x processors by setting `CBUFEN` on `MODE1`

Scratch Registers

The `cc21k` C/C++ run-time environment specifies a set of registers whose contents do not need to be saved and restored. Note that the contents of these registers are not preserved across function calls. Table 1-18 lists these registers.

Table 1-18. Scratch Registers

b4	b12	b13	r0	r1	r2	r4	r8	r12
i4	i12	m4	m12	i13	PX	USTAT1	USTAT2	

In addition, for ADSP-2116x processors, the `PEy` data registers are all scratch registers. Table 1-19 lists these registers.

The USTAT registers are now treated as scratch registers.

Table 1-19. Additional ADSP-2116x Scratch Registers

s0	s1	s2	s3	s4	s5	s6	s7	s8
s9	s10	s11	s12	s13	s14	s15	USTAT3	USTAT4
ASTATy	STKy							

Stack Registers

The `cc21k` C/C++ run-time environment reserves a set of registers for controlling the run-time stack. These registers may be modified for stack management, but they must be saved and restored. Table 1-20 lists these registers.

Table 1-20. Pointer Registers

Register	Value	Modification Rules
i7	Stack pointer	Modify for stack management, restore when done
i6	Frame pointer	Modify for stack management, restore when done
i12	Return address	Load with function call return address on function exit

Alternate Registers

The C/C++ run-time environment model does not use any of the alternate registers because these registers are available for use in assembly language only. To use these registers, several aspects of the C/C++ run-time model must be understood.

The C/C++ run-time model uses register `I6` as the frame pointer and register `I7` as the stack pointer. Setting the `DAG` register that contains `I6` and `I7` from a background register to an active register directly affects the stack operation. The C/C++ run-time model does not have an understanding of background registers.

If the background I6 and I7 registers are active and an interrupt occurs, the C/C++ run-time model still uses I6 and I7 to update the stack. This results in faulty stack handling.

The background register set containing DAG registers I6 and I7 should only be used in assembly routines if interrupts are not enabled.

The super fast interrupt dispatcher uses context switching rather than saving registers on the run-time stack. To ensure no register conflicts, do not use the super fast interrupt dispatcher or disable interrupts when using secondary registers in an assembly routine.

Managing the Stack

The cc21k C/C++ run-time environment uses the run-time stack for storage of automatic variables and return addresses. The stack is managed by a Frame Pointer (FP) and a Stack Pointer (SP) and grows downward in memory, moving from higher to lower addresses.

A stack frame is a section of the stack used to hold information about the current context of the C or C++ program. Information in the frame includes local variables, compiler temporaries, and parameters for the next function.

The Frame Pointer serves as a base for accessing memory in the stack frame. Routines refer to locals, temporaries, and parameters by their offset from the frame pointer.

Figure 1-4 shows an example section of a run-time stack. In the figure, the currently executing routine, Current(), was called by Previous(), and Current() in turn calls Next(). The state of the stack is as if Current() has pushed all the arguments for Next() onto the stack and is just about to call Next().

Stack usage for passing any or all of a function's arguments depends on the number and types of parameters to the function.

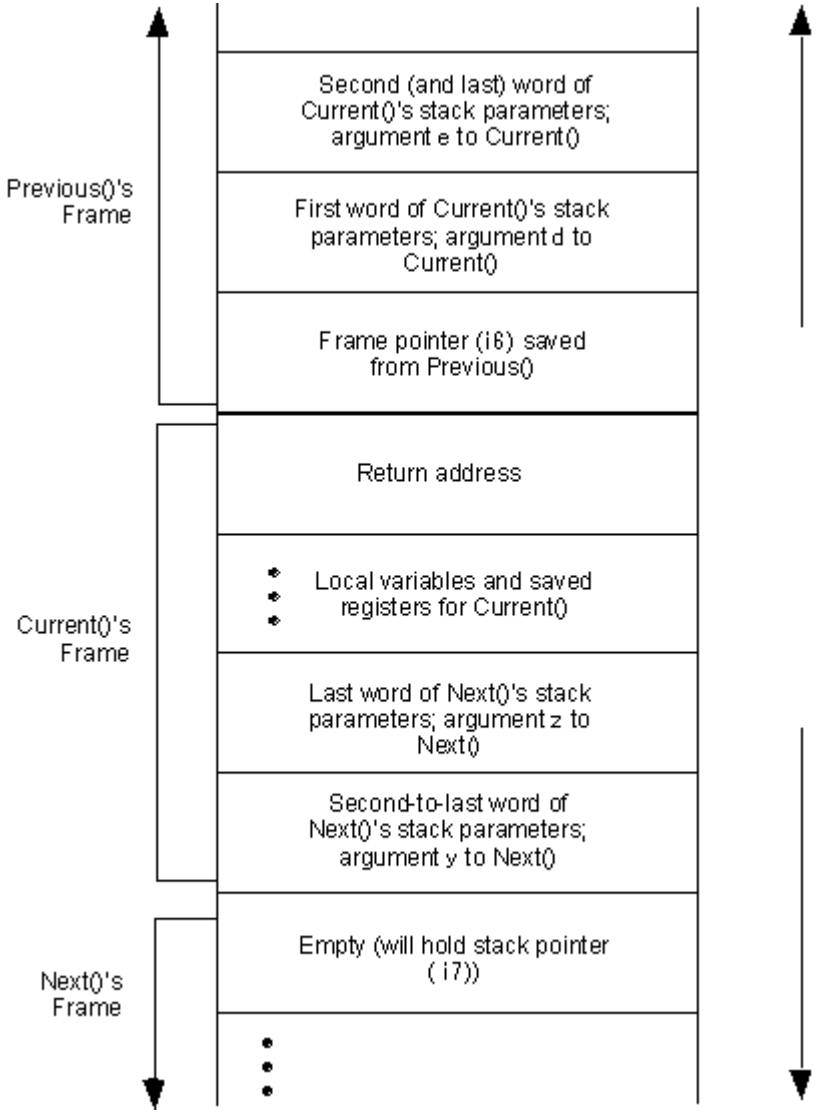


Figure 1-4. Example Run-Time Stack

The prototypes for the functions in Figure 1-4 are as follows:

```
void Current(int a, int b, int c, int d, int e);
void Next(int v, int w, int x, int y, int z);
```

In generating code for a function call, the compiler produces the following operations to create the called function's new stack frame:

- Loads the `r2` register with the frame pointer (in the `i6` register)
- Sets the FP, `i6` register, equal to the SP (in the `i7` register)
- Uses the delayed-branch instruction to pass control to the called function
- Pushes the FP, `r2`, onto the run-time stack during the first branch delay slot
- Pushes the return address, `pc`, onto the run-time stack during the second delay-branch slot

For the ADSP-21020 processor, the following instructions create a new stack frame.

```
r2=i6;
i6=i7;
jump my_function (DB);
      /* where my_function is the called function */
dm(i7, m7) = r2;
dm(i7, m7) = pc;
```

For ADSP-2106x/2116x/2126x/2136x processors, the following instructions create a new stack frame. Note how the two initial register moves are incorporated into the `cjump` instruction.

```
cjump my_function (DB);
      /* where my_function is the called function */
dm(i7, m7) = r2;
dm(i7, m7) = pc;
```

As you write assembly routines, note that the operations to create a stack frame are the responsibility of the called function, and you can use the `entry` or `leaf_entry` macros to perform these operations. For more information on these macros, see "Using Mixed C/C++ and Assembly Support Macros" on page 1-214.

In generating code for a function return, the compiler uses the following operations to restore the calling function's stack frame.

- Pops the return address off the run-time stack and loads it into the `i12` register
- Uses the delayed-branch instruction to pass control to the calling function and jumps to the return address (`i12 + 1`)
- Restores the caller's stack pointer, `i7` register, by setting it equal to FP, `i6` register, during the first branch delay slot
- Restores the caller's frame pointer, `i6` register, by popping the previously saved FP off the run-time stack and loading the value into `i6` during the second delay-branch slot

For ADSP-2106x/2116x/2126x/2136x processors, the following instructions return from the function and restore the stack and frame pointers. Note that the restoring of SP and FP are incorporated into the `rframe` instruction.

```
i12 = dm(-1, i6);
jump (m14, i12) (DB);
nop;
rframe;
```

As you write assembly routines, note that the operations to restore stack and frame pointers are the responsibility of the called function, and you can use the `exit` or `leaf_exit` macros to perform these operations. For more information on these macros, see "Using Mixed C/C++ and Assembly Support Macros" on page 1-214.

In the following code examples (Listing 1-1 and Listing 1-2), observe how the function calls in the C code translate to stack management tasks in the compiled (assembly) version of the code. The comments have been added to the compiled code to indicate the function prologue and function epilogue.

Listing 1-1. Stack Management, Example C Code

```
/* Stack management — C code */

int my_func(int, int);
int arg_a, return_c;

main()
{
   static int arg_b;
   arg_b = 0;
   return_c = my_func(arg_a, arg_b);
}

int my_func(int arg_1, int arg_2);
{
   return (arg_1 + arg_2)/2;
}
```

Listing 1-2. Stack Management, Example ADSP-2106x Assembly Code

```
/* Stack management — C compiled (2106x assembly) code */
.section /pm seg_pmco;
.global _main;
_main:
      .def end_prologue; .val .; .scl 109; .endef;
   r4=dm(_arg_a);
   /* r4, the first argument register, which is arg_a */
   r8=0;
   /* r8, the second argument register, which is arg_b */
   dm(arg_b)=r8;

/* The next three lines are the function call sequence */
   cjump (pc,_my_func) (DB);
```

```
    dm(i7,m7)=r2;
    dm(i7,m7)=pc;

    dm(_return_c)=r0;

    /* The next four lines are main's function epilogue */
    i12=dm(-1,i6);
    jump (m14, i12) (DB);
    nop;
    rframe;

.global _my_func;
_my_func:
    .def end_prologue; .val .; .scl 109; .endef;
    r0=(r4+r8)/2;

    /* The next four lines are my_func's function epilogue */
    i12=dm(-1,i6);
    jump (m14, i12) (DB);
    nop;
    rframe;
```

The next two sections, "Transferring Function Arguments and Return Value" on page 1-202 and "Using Macros to Manage the Stack" on page 1-228, provide additional detail on function call requirements.

Transferring Function Arguments and Return Value

The C/C++ run-time environment uses a set of registers and the run-time stack to transfer function parameters to assembly routines. Your assembly language functions must follow these conventions when they call or when they are called by C or C++ functions.

Because it is most efficient to use registers for passing parameters, the run-time environment attempts to pass the first three parameters in a function call using registers; it then passes any remaining parameters on the run-time stack.

The convention is to pass the function's first parameter in r4, the second parameter in r8, and the third parameter in r12. The following exceptions apply to this convention:

- If any parameter is larger then a single 32-bit word, then that parameter and all subsequent parameters are passed on the stack.
- If the function is declared to take a variable number of arguments (has ... in its prototype), then the last named parameter and any subsequent parameters are passed on the stack.

Table 1-21 lists the rules that cc21k uses for passing parameters in registers to functions and the rules that your assembly code must use for returns.

Table 1-21. Parameter and Return Value Transfer Registers

Register	Parameter Type Passed Or Returned
r4	Pass first 32-bit data type parameter
r8	Pass second 32-bit data type parameter
r12	Pass third 32-bit data type parameter
stack	Pass fourth and remaining parameters; see exceptions to this rule on page 1-203
r0	Return int, long, char, float, short, pointer, and one-word structure parameters
r0, r1	Return long double and two-word structure parameters. Place MSW in r0 and LSW in r1
r1	Return the address of results that are longer than two words; r1 contains the first location in the block of memory containing the results

Consider the following function prototype example.

```
pass(int a, float b, char c, float d);
```

The first three arguments, a, b, and c are passed in registers r4, r8, and r12, respectively. The fourth argument, d, is passed on the stack.

This next example illustrates the effects of passing `doubles`.

```
count(int w, long double x, char y, float z);
```

The first argument, `w`, is passed in `r4`. Because the second argument, `x`, is a multi-word argument, `x` is passed on the stack. As a result, the remaining arguments, `y` and `z`, are also passed on the stack.

The following example illustrates the effects of variable arguments on parameter passing.

```
compute(float k, int l, char m,…);
```

Here, the first two arguments, `k` and `l`, are passed in registers `r4` and `r8`. Because `m` is the last named argument, `m` is passed on the stack, as are all remaining variable arguments.

When arguments are placed on the stack, they are pushed on from right to left. The right-most argument is at a higher address than the left-most argument passed on the stack.

The following example shows how to access parameters passed on the stack.

```
tab(int a, char b, float c, int d, int e, long double f);
```

Parameters `a`, `b`, and `c` are passed in registers because they are single-word parameters. The remaining parameters, `d`, `e`, and `f`, are passed on the stack.

All parameters passed on the stack are accessed relative to the frame pointer, register `i6`. The first parameter passed on the stack, `d`, is at address `i6 + 1`. To access it, you could use this assembly language statement.

```
r3=dm(1,i6);
```

The second parameter passed on the stack, `e`, is at `i6 + 2` and can be accessed by the statement

```
r3=dm(2,i6);
```

The third parameter passed on the stack, `f`, is a `long double` that has its most significant word at `i6 + 3` and its least significant word at `i6 + 4`. The most significant word of `f` can be accessed by the statement

```
r3=dm(3,i6);
```

Using Data Storage Formats

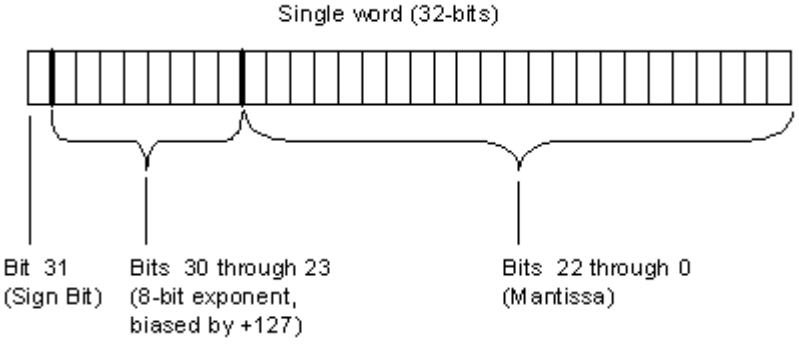
The C/C++ run-time environment uses the data formats that appear in the Table 1-22, Table 1-23, Figure 1-5, and Figure 1-6 on page 1-207.

Table 1-22. Data Storage Formats and Data Type Sizes

Applied Type	Number Representation
`int`	32-bit two's complement
`long int`	32-bit two's complement
`short int`	32-bit two's complement
`unsigned int`	32-bit unsigned magnitude
`unsigned long int`	32-bit unsigned magnitude
`char`	32-bit two's complement
`unsigned char`	32-bit unsigned magnitude
`float`	32-bit IEEE single-precision
`double`	32-bit IEEE single-precision or 64-bit IEEE double-precision if you compile with the `-double-size-64` switch
`long double`	64-bit IEEE double-precision

Table 1-23. Data Storage Formats and Data Storage

Data	Big Endian Storage Format
`long double`	Writes 64-bit IEEE double-precision data with the most significant word closer to address `0x0000`, proceeds toward the top of memory with the rest (see Figure 1-6 on page 1-207 for details).



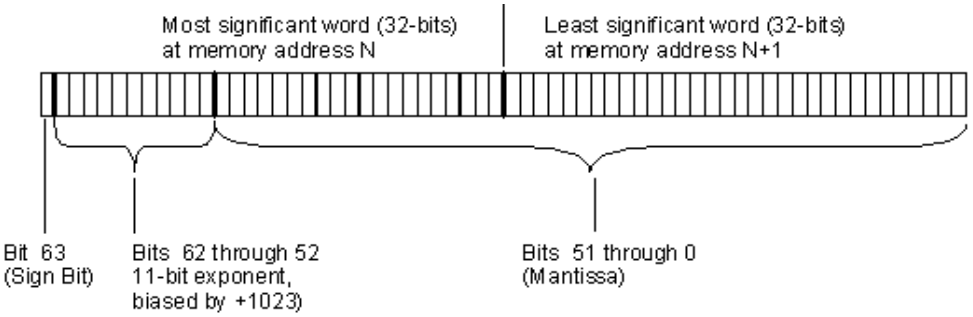
The single word (32-bit) data storage format equates to:

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

Where:

Sign	Comes from the Sign Bit
Mantissa	Represents the fractional part of the Mantissa (23-bits). The 1. is assumed in this format.
Exponent	Represents the 8-bit exponent

Figure 1-5. Floating-Point (32-Bit IEEE Single-Precision) Storage



The two word (64-bit) data storage format equates to:

$$-1^{Sign} \times 1.Mantissa \times 2^{(Exponent - 1023)}$$

Where:

Sign	Comes from the Sign Bit
Mantissa	Represents the fractional part of the Mantissa (52-bits). The 1. is assumed in this format.
Exponent	Represents 11-bit exponent

Figure 1-6. Floating-Point (64-Bit IEEE Double-Precision) Storage

Using the Run-Time Header

The run-time header is an assembly language procedure that initializes the processor and sets up processor features to support the C/C++ run-time environment. The source code for the default run-time headers is in:

- `020_hdr.asm` for ADSP-21020 processors
- `06x_hdr.asm` for ADSP-2106x processors
- `16x_hdr.asm` for ADSP-2116x processors
- `26x_hdr.asm` for ADSP-2126x processors
- `36x_hdr.asm` for ADSP-2316x processors

This run-time header performs the following operations:

- Initializes the C/C++ run-time environment
- Sets up the interrupt table
- Calls your `main()` routine

C/C++ and Assembly Interface

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

Before attempting to do 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) in "C/C++ Run-Time Environment" on page 1-175. At the end of this reference, a series of examples demonstrate how to mix C/C++ and assembly code.

Calling Assembly Subroutines from C/C++ Programs

Before calling an assembly language subroutine from a C or C++ program, you should create a prototype to define the arguments for the assembly language subroutine and the interface from the C or C++ program to the assembly language subroutine. You can legally use a function without a prototype in C. However, using prototypes is strongly recommended for good software engineering. When the prototype is omitted, the compiler cannot do argument type-checking and assumes that the return value is of type integer.

The compiler prefaces the name of any external entry point with an underscore. You should either declare your assembly language subroutine's name with a leading underscore or define it within an `'extern "asm" {}'` format to tell the compiler that it is an assembly language subroutine.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated* registers. The scratch registers can be used within the assembly language program without worrying about their previous contents. If you need more room (or are working with existing code) and

wish to use the preserved registers, you must first save their contents and then restore those contents before returning. 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.

The compiler also assumes that the machine state does not change during execution of the assembly language subroutine.

Do not change any machine modes (for example, the machine may have an integer/fractional mode, or it may use certain registers to indicate circular buffering when those register values are non-zero).

If arguments are on the stack, they are addressed via an offset from the stack pointer or frame pointer.

A good way to explore how arguments are passed between a C/C++ program and an assembly language subroutine is to write a dummy function in C/C++ and compile it with the `save temporary files` option (the `-save-temps` command-line option). The following example includes the global volatile variable assignments to indicate where the arguments can be found upon entry to `asmfunc`.

```
// Sample file for exploring compiler interface …
   // global variables … assign arguments there just so
   // we can track which registers were used
   // (type of each variable corresponds to one of arguments):

   int global_a;
   float global_b;
   int * global_p;

// the function itself:

int asmfunc(int a, float b, int * p);
{
   // do some assignments so .s file will show where args are:
   global_a = a;
```

```
    global_b = b;
    global_p = p;

    // value gets loaded into the return register:
    return 12345;
}
```

When compiled with the `-save-temps -O0` option set, this produces the following code.

```
// PROCEDURE: asmfunc
.global _asmfunc;
_asmfunc:
       modify(i7,-7);
       dm(-8,i6)=r3;
       dm(-7,i6)=r6;
       r2=i0;
       dm(-6,i6)=r2;
       dm(-4,i6)=r4;
       dm(-3,i6)=r8;
       dm(-2,i6)=r12;
       r3=r4;
       r6=r8;
       i0=r12;
       dm(_global_a)=r3;
       dm(_global_b)=r6;
       r2=i0;
       dm(_global_p)=r2;
       r0=12345;
```

Calling C/C++ Functions from Assembly Programs

You may want to call C or C++-callable library and other functions from within an assembly language program. As discussed in "Calling Assembly Subroutines from C/C++ Programs" on page 1-209, you may wish to create a test function to do this in C or 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* first *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 are not changed by calling a C/C++ function. The function always saves and restores the contents of preserved registers if they are going to change.

If arguments are on the stack, they are addressed via an offset from the stack pointer or frame pointer. You can explore how arguments are passed between an assembly language program and a function by writing a dummy function in C or C++ and compiling it with the `save temporary files` option (the `-save-temps` command line option). 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 or C++ main program to initialize the run-time system; hold the stack until it is needed by the C/C++ function being called from the assembly language program; and then hold that stack until it is needed to call back into C/C++, making 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 callee.

The following example demonstrates the features described in this section. Because so many different features are combined into a single example, this procedure as a whole should not be viewed as an example of good assembly programming.

```
// PROCEDURE: memalloc
.global _memalloc;
_memalloc:
     r5=0xffff;        // Assign a value to preserved reg r5
     r8=0xffff;        // Assign a value to scratch reg r8

     r0=dm(-3,i6);     // Read a value from the stack
     r4=r0;            // Put this value in a parameter register

     // Save value of scratch register prior to function call
     r7=r8;

     // Call the C function malloc()
     r2=i6;
     i6=i7;
     jump _malloc (DB);
     dm(i7,m7)=r2;
     dm(i7,m7)=pc;

     // Check the result of the function call
     r0=pass r0;
     if eq jump(pc,_error);

     // Check that the preserved register did not change over
     // the function call
     r4=0xffff;
     comp(r4,r5);
     if ne jump(pc, _error);

     // Restore value of scratch register after function call
     r8=r7;

     i6 = 0x123;        // PROGRAMMING ERROR! Do not change
                        // dedicated registers
     rts;
```

Using Mixed C/C++ and Assembly Support Macros

This section lists C/C++ and assembly interface support macros in the `asm_sprt.h` system header file. Use these macros for interfacing assembly language modules with C or C++ functions. Table 1-24 lists the macros.

Although the syntax for each macro does not change, the listing of `asm_sprt.h` in this section may not be the most recent version. To see the current version, check the `asm_sprt.h` file that came with your software package.

Table 1-24. Interface Support Macro Summary

`entry`	`exit`	`leaf_entry`	`leaf_exit`
`ccall(x)`	`reads(x)`	`puts`	`gets(x)`
`alter(x)`	`save_reg`	`restore_reg`	

The following are the descriptions and the syntax for the C/C++ and assembly interface support macros.

entry

The `entry` macro expands into the function prologue for non-leaf functions. This macro should be the first line of any non-leaf assembly routine. Note that this macro is currently null, but it should be used for future compatibility.

exit

The `exit` macro expands into the function epilogue for non-leaf functions. This macro should be the last line of any non-leaf assembly routine. Exit is responsible for restoring the caller's stack and frame pointers and jumping to the return address. Note that this macro is currently null, but it should be used for future compatibility.

leaf_entry

The `leaf_entry` macro expands into the function prologue for leaf functions. This macro should be the first line of any non-leaf assembly routine. Note that this macro is currently null, but it should be used for future compatibility.

leaf_exit

The `leaf_exit` macro expands into the function epilogue for leaf functions. This macro should be the last line of any leaf assembly routine. `leaf_exit` is responsible for restoring the caller's stack and frame pointers and jumping to the return address.

ccall(x)

The `ccall` macro expands into a series of instructions that save the caller's stack and frame pointers and then jump to function `x()`.

reads(x)

The `reads` macro expands into an instruction that reads a value from the stack. The value is located at an offset `x` from the frame pointer.

puts=x

The `puts` macro expands into an instruction that pushes the value in register `x` onto the stack.

gets(x)

The `gets` macro expands into an instruction that pops a value off of the stack and puts the value in the indicated register:

```
register = gets(x);
```

The value is located at an offset `x` from the stack pointer.

alter(x)

The `alter` macro expands into an instruction that adjusts the stack pointer by adding the immediate value `x`. With a positive value for `x`, `alter` pops x words from the top of the stack. You could use `alter` to clear `x` number of parameters off the stack after a call.

save_reg

The `save_reg` macro expands into a series of instructions that push the register file registers (`r0-r15`) onto the run-time stack.

restore_reg

The `restore_reg` macro expands into a series of instructions that pop the register file registers (`r0-r15`) off the run-time stack.

The following code example shows the `asm_sprt.h` used for defining C/C++/assembly interface support macros.

```
/* asm_sprt.h — C/C++/Assembly Interface Support Macros */
/* asm_sprt.h - $Date: 10/09/97 6:28p $                 */

#ifndef __ASM_SPRT_DEFINED
#define __ASM_SPRT_DEFINED

#define entry                   /* nothing */
#define leaf_entry              /* nothing */

#ifdef __ADSP21020__
#define ccall(x) \
   r2=i6; i6=i7; \
   jump (pc, x) (db); \
   dm(i7,m7)=r2; \
   dm(i7,m7)=PC;
#define leaf_exit \
   i12=dm(m7,i6); \
   jump (m14,i12) (db); \
   i7=i6; i6=dm(0,I6);
```

```
#define exit leaf_exit
#else
#define ccall(x) \
   cjump (x) (DB); \
   dm(i7,m7)=r2; \
   dm(i7,m7)=PC;

#define leaf_exit \
   i12=dm(m7,i6); \
   jump (m14,i12) (db); \
   nop; \
      RFRAME
#define exit leaf_exit
#endif

#define reads(x)dm(x, i6)
#define putsdm(i7, m7)
#define gets(x)dm(x, i7)
#define alter(x)modify(i7, x)

#define save_reg \
         puts=r0;\
         puts=r1;\
         puts=r2;\
         puts=r3;\
         puts=r4;\
         puts=r5;\
         puts=r6;\
         puts=r7;\
         puts=r8;\
         puts=r9;\
         puts=r10;\
         puts=r11;\
         puts=r12;\
         puts=r13;\
         puts=r14;\
         puts=r15;

#define restore_reg \
         r15=gets(1);\
```

```
        r14=gets(2);\
        r13=gets(3);\
        r12=gets(4);\
        r11=gets(5);\
        r10=gets(6);\
        r9 =gets(7);\
        r8 =gets(8);\
        r7 =gets(9);\
        r6 =gets(10);\
        r5 =gets(11);\
        r4 =gets(12);\
        r3 =gets(13);\
        r2 =gets(14);\
        r1 =gets(15);\
        r0 =gets(16);\
        alter(16);

#endif
```

Using Mixed C/C++ and Assembly Naming Conventions

It is necessary to be able to use C or C++ symbols (function or variable names) in assembly routines and use assembly symbols in C or C++ code. This section describes how to name and use C/C++ and assembly symbols.

To name an assembly symbol that corresponds to a C or 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 function or variable in an assembly routine, declare it as global in the C program. Import the symbol into the assembly routine by declaring the symbol with the `.EXTERN` assembler directive.

The external name of a C++ function encodes information about its type and parameters. Function “signature” enables the overloading of functions and operators that the C++ language supports. To reference a function in

a C++ module, declare it with the `extern "C"` specifier in order to use the naming convention of C. Note that C++ data symbols use the same convention as C.

To use an assembly function or variable in your C program, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern` in the C program.

To use an assembly function in your C++ module, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern "C"` in the C++ program. For example, to reference the `_funcmult` assembly routine from a C++ program, you declare it as `extern "C" int funcmult(int a, int b)` in the C++ program.

Table 1-25 shows several examples of the C/Assembly interface naming conventions. Each row shows how assembler code can reference the corresponding C item.

Table 1-25. C Naming Conventions For Symbols

In the C Program	In the Assembly Subroutine
`int c_var; /*declared global*/`	`.extern _c_var;`
`void c_func(){...}`	`.extern _c_func;`
`extern int asm_var;`	`.global _asm_var;`
`extern void asm_func();`	`.global _asm_func;` `_asm_func:`

on page 1-220 shows several examples of the C++/Assembly interface naming conventions. Each row shows how assembler code can reference the corresponding C++ item.

Table 1-26. C++ Naming Conventions for Symbols

In the C++ Program	In the Assembly Subroutine
`extern "C" { int c_var; } /*declared global*/`	`.extern _c_var;`
`extern "C" void c_func(void){...}`	`.extern _c_func;`
`extern "C" int asm_var;`	`.global _asm_var;`
`extern "C" void asm_func(void);`	`.global _asm_func;` `_asm_func:`

Implementing C++ Member Functions in Assembly Language

If an assembly language implementation is desired for a C++ member function, the simplest way is to use C++ to provide the proper interface between C++ and assembly.

In the class definition, write a simple member function to call the assembly-implemented function (subroutine). This call can establish any interface between C++ and assembly, including passing a pointer to the class instance. Since the call to the assembly subroutine resides in the class definition, the compiler inlines the call (inlining adds no overhead to compiler performance). From an efficiency point of view, the assembly language function is called directly from the user code.

As for any C++ function, ensure that a prototype for the assembly-implemented function is included in your program. As discussed in "Using Mixed C/C++ and Assembly Naming Conventions" on page 1-218, you declare your assembly language subroutine's name with the `.GLOBAL` directive in the assembly portion and import the symbol by declaring it as `extern "C"` in the C++ portion of the code.

Note that using this method you avoid name mangling—you choose your own identifier for the external function. Access to internal class information can be done either in the C++ portion or in the assembly portion. If the assembly subroutine needs only limited access to the class members, it

is easier to select those in the C++ code and pass them as explicit arguments. This way the assembly code does not need to know how data is allocated within a class instance.

```
#include <stdio.h>
/* Prototype for external assembly routine: */
/* C linkage does not have name mangling */
extern "C" int cc_array(int);

class CC {
      private:
         int av;
      public:
         CC(){};
         CC(int v) : av(v){};
         int a() {return av;};
                                 /* Assembly routine call: */
         int array() {return cc_array(av);};
};

int main()
{
      CC samples(11);
      CC points;
      points = CC(22);
      int j, k;
      j = samples.a();
      k = points.array();     // Test asm call

      printf ( "Val is %d\n", j );
      printf ( "Array is %d\n", k );

      return 1;
}
/* In a separate assembly file: */
      .section /pm seg_pmco;
      .global _cc_array;
      _cc_array:
      modify(i7,-3);
      dm(-4,i6)=r3;
      dm(-2,i6)=r4;
      r3=r4;
      r0=r3+r3;
```

```
r3=dm(-4,i6);
i12=dm(m7,i6);
jump(m14,i12)(DB);
rframe;
nop;
```

Writing C/C++ Callable SIMD Subroutines

You can write assembly subroutines that use SIMD mode for the ADSP-2116x, ADSP-2126x and ADSP-2136x processors and call them from your C programs. The routine may use SIMD mode (`PEYEN` bit=1) for all code between the function prologue and epilogue, placing the chip in SISD mode (`PEYEN` bit =0) before the function epilogue or returning from the function.

While it is possible to write subroutines that can be called in SIMD mode (the chip is in SIMD mode before the call and after the return), the compiler does not support a SIMD call interface at this time. For example, trying to call a subroutine from a `#pragma SIMD_for` loop prevents the compiler from executing the loop in SIMD mode because the compiler does not support SIMD mode calls.

Because transfers between memory and data registers are doubled in SIMD mode (each explicit transfer has a matching implicit transfer), it is recommended that you access the stack in SISD mode to prevent corrupting the stack. For more information on SIMD mode memory accesses, see the *Memory* chapter in the hardware reference for an appropriate ADSP-2116x processors.

If you are using SIMD subroutines, your interrupt handler must provide additional support. This support in the interrupt service routine entails saving-restoring the `PEYEN` bit and placing the processor in the mode (SISD or SIMD) that the interrupt service routine needs. Interrupt handlers often use the `MMASK` register to expedite these mode changes.

C++ Programming Examples

This section provides the following examples for C++-specific features:

- "Using Fract Support" on page 1-223
- "Using Complex Support" on page 1-224

Note that the `cc21k` compiler runs in C mode by default. To run the compiler in C++ mode, select the corresponding option on the command line, or check it in the **Project Options** dialog box of the VisualDSP++ environment.

For example, the following command line

```
cc21k -c++ fdot.c -T060.ldf
```

runs `cc21k` with:

`-c++`	Specifies that the following source file is written in ANSI/ISO standard C++ extended with the Analog Devices keywords.
`fdot.c`	Specifies the source file for your program.
`-T 060.ldf`	Specifies the Linker Description File for the ADSP-21060 processors.

Using Fract Support

Listing 1-3 on page 1-224 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.

For more information about the fractional data type and arithmetic, see "C++ Fractional Type Support" on page 1-146.

Listing 1-3. Dot Product Using Fract Arithmetic Example — C++ Code

```
fract fdot (int array_size, fract *x, fract *y)
   {
      int j;
      fract s;
      s = 0;
      for (j=0; j < array_size; j++)
         {
            s += x[j] * y[j];
         }
      return s;
   }
int main(void)
   {
      set_saturate_mode();
      fdot (N,x,y);
   }
```

Using Complex Support

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

```
z := z * z + c
```

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

Listing 1-4 demonstrates the Mandelbrot generator expressed in a simple algorithm using the C++ library `complex` class.

Listing 1-4. 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 1-5 shows a C version of the inner computational function of the Mandelbrot generator, which extracts performance and programming penalties (compared with the C++ version).

Listing 1-5. 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);
}
```

Mixed C/C++/Assembly Programming Examples

This section shows examples of types of mixed C/C++/assembly programming in order of increasing complexity. The examples in this section are as follows:

- "Using Inline Assembly (Add)" on page 1-227
- "Using Macros to Manage the Stack" on page 1-228
- "Using Scratch Registers (Dot Product)" on page 1-229
- "Using Void Functions (Delay)" on page 1-231
- "Using the Stack for Arguments and Return (Add 5)" on page 1-232
- "Using Registers for Arguments and Return (Add 2)" on page 1-233
- "Using Non-Leaf Routines That Make Calls (RMS)" on page 1-234
- "Using Call Preserved Registers (Pass Array)" on page 1-236

Note that leaf assembly routines are routines that return without making any calls. Non-leaf assembly routines call other routines before returning to the caller.

Note that you can use `cc21k` to compile your C or C++ program and assemble your assembly language modules. This ensures that the assembly of your modules complies with the C/C++ run-time environment.

For example, the following `cc21k` command line

```
cc21k my_prog.c my_sub1.asm -T 062.ldf -Wremarks
```

runs `cc21k` with:

`my_prog.c`	Selects a C language source file for your program
`my_sub1.asm`	Selects an assembly language module to be assembled and linked with your program
`-T 062.ldf`	Selects a Linker Description File describing your system
`-Wremarks`	Selects diagnostic compiler warnings

Using Inline Assembly (Add)

The following example shows how to write a simple routine in ADSP-21xxx assembly code that properly interfaces to the C/C++ environment. It uses the `asm()` construct to pass inline assembly code to the compiler.

```
int i,j,k,l;
main()
{
   l = add(i,j,k);
}

/* Per the run-time environment, function add() passes arg x in
   r4, arg y in r8, and arg z in r12. Then, func adds args and
   puts return in r0. */

add(int x, int y, int z)
{
   asm("r0=%0+%1; r0=r0+%2"::"d"(x),"d"(y),"d"(z));
}
```

Using Macros to Manage the Stack

Listing 1-6 and Listing 1-7 on page 1-229 show how macros can simplify function calls between C, C++, and assembly functions. The assembly function uses the `entry`, `exit`, and `ccall` macros to keep track of return addresses and manage the run-time stack. For more information, see "Managing the Stack" on page 1-197.

Listing 1-6. Subroutine Return Address Example — C Code

```
/* Subroutine Return Address Example—C code: */
/* assembly and c functions prototyped here  */
void asm_func(void);
void c_func(void);

                     /* c_var defined here as a global */
                     /* used in .asm file as _c_var    */
int c_var=10;

           /* asm_var defined in .asm file as _asm_var */
extern int asm_var;

main ()
{
   asm_func();       /* call to assembly function       */
}
              /* this function gets called from asm file */
void c_func(void)
{
   if (c_var != asm_var)
      exit(1);
   else
      exit(0);
}
```

Listing 1-7. Subroutine Return Address Example—Assembly Code

```
/* Subroutine Return Address Example—Assembly code:   */
#include <asm_sprt.h>
.section/dm seg_dmda;
.var _asm_var=0;          /* asm_var is defined here   */
.global _asm_var;         /* global for the C function */

.section/pm seg_pmco;
.global _asm_func;        /* _asm_func is defined here */
.extern _c_func;          /* c_func from the C file    */
.extern _c_var;           /* c_var from the C file     */

_asm_func:
entry;                    /* entry macro from asm_sprt */

   r8=dm(_c_var);         /* access the global C var   */
   dm(_asm_var)=r8;       /* set _asm_var to _c_var)   */

      ccall(_c_func);     /* call the C function       */

      exit;               /* exit macro from asm_sprt  */
```

Using Scratch Registers (Dot Product)

To write assembly functions that can be called from a C or C++ program, your assembly code must follow the conventions of the C/C++ run-time environment and use the conventions for naming functions. The `dot()` assembly function described below demonstrates how to comply with these specifications.

This function computes the dot product of two vectors. The two vectors and their lengths are passed as arguments. Because the function uses only scratch registers (as defined by the run-time environment) for intermediate values and takes advantage of indirect addressing, the function does not need to save or restore any registers.

```
/* dot(int n, dm float *x, pm float *y);
Computes the dot product of two floating-point vectors of length
n. One is stored in dm and the other in pm. Length n must be
greater than 2.*/

#include <asm_sprt.h>

.section/pm seg_pmco;
/* By convention, the assembly function name is the C function
name with a leading underscore; "dot()" in C becomes "_dot" in
assembly */

.global _dot;
_dot:

leaf_ entry;

r0=r4-1,i4=r8;
/* Load first vector address into I register, and load r0 with
length -1 */

r0=r0-1,i12=r12;
/* Load second vector address into I register and load r0 with
length-2 (because the 2 iterations outside feed and drain the
pipe */

f12=f12-f12,f2=dm(i4,m6),f4=pm(i12,m14);
/* Zero the register that will hold the result and start
feeding pipe */

f8=f2*f4, f2=dm(i4,m6),f4=pm(i12,m14);
/* Second data set into pipeline, also do first multiply */

lcntr=r0, do dot_loop until lce;
/* Loop length-2 times, three-stage pipeline: read, mult, add */

dot_loop:
   f8=f2*f4, f12=f8+f12,f2=dm(i4,m6),f4=pm(i12,m14);
   f8=f2*f4, f12=f8+f12;
   f0=f8+f12;
```

```
/* drain the pipe and end with the result in r0, where it'll be
returned */

leaf_exit;
/* restore the old frame pointer and return */
```

Using Void Functions (Delay)

The simplest kind of assembly routine is one with no arguments and no return value, which corresponds to C/C++ functions prototyped as `void my_function(void)`. Such routines could be used to monitor an external event or used to perform an operation on a global variable.

It is important when writing such assembly routines to pay close attention to register usage. If the routine uses any call-preserved or compiler reserved registers (as defined in the run-time environment), the routine must save the register and restore it before returning. Because the following example does not need many registers, this routine uses only scratch registers (also defined in the run-time environment) that do not need to be saved.

Note that in the example all symbols that need to be accessed from C/C++ contain a leading underscore. Because the assembly routine name `_delay` and the global variable `_del_cycle` must both be available to C and C++ programs, they contain a leading underscore in the assembly code.

```
/* Simple Assembly Routines Example — _delay */
/* void delay (void);
An assembly language subroutine to delay N cycles, where N is
the value of the global variable del_cycle */

#include <asm_sprt.h>;

.section/pm seg_pmco;
.extern _del_cycle;
.global _delay;
_delay:
   leaf_entry;                  /* first line of any leaf func */
```

```
   R4 = DM (_del_cycle);
/* Here, register r4 is used because it is a scratch register
and does not need to be preserved */
   LCNTR = R4, DO d_loop UNTIL LCE;
   d_loop:
   nop;
   leaf_exit;                    /* last line of any leaf func */
```

Using the Stack for Arguments and Return (Add 5)

A more complicated kind of routine is one that has parameters but no return values. The following example adds together the five integers passed as parameters to the function.

```
/* Assembly Routines With Parameters Example — _add5 */
/* void add5 (int a, int b, int c, int d, int e);
An assembly language subroutine that adds 5 numbers */

#include <asm_sprt.h>
.section/pm seg_pmco;
.extern _sum_of_5;   /* variable where sum will be stored */
.global _add5;

_add5:
leaf_entry;
/* the calling routine passes the first three parameters in
registers r4, r8, r12 */

r4 = r4 + r8;      /* add the first and second parameter */
r4 = r4 + r12;     /* adds the third parameter           */

/* the calling routine places the remaining parameters
(fourth/fifth) on the run-time stack; these parameters can be
accessed using the reads() macro */

r8 = reads(1);     /* put the fourth parameter in r8     */
r4 = r4 + r8;      /* adds the fourth parameter          */
r8 = reads(2);     /* put the fifth parameter in r8      */
r4 = r4 + r8;      /* adds the fifth parameter           */
```

```
dm(_sum_of_5) = r4;
/* place the answer in the global variable */

leaf_exit;
```

Using Registers for Arguments and Return (Add 2)

There is another class of assembly routines in which the routines have both parameters and return values. The following example of such a routine adds two numbers and returns the sum. Note that this routine follows the run-time environment specification for passing function parameters (in registers `r4` and `r8`) and passing the return value (in register `r0`).

```
/* Routine With Parameters & Return Value —add2_ */
/* int add2 (int a, int b);
An assembly language subroutine that adds two numbers and returns
the sum */

#include <asm_sprt.h>

.section/pm seg_pmco;
.global _add2;
_add2:
leaf_entry;

/* per the run-time environment, the calling routine passes the
first two parameters passed in registers r4 and r8; the return
value goes in register r0 */

r0 = r4 + r8;
/* add the first and second parameter, store in r0*/

leaf_exit;
```

Using Non-Leaf Routines That Make Calls (RMS)

A more complicated example, which calls another routine, computes the root mean square of two floating-point numbers, such as

$$z = \sqrt{x^2 + y^2}$$

Although it is straight forward to develop your own function that calculates a square-root in ADSP-21xxx assembly language, the following example demonstrates how to call the square root function from the C/C++ run-time library, `sqrtf`. In addition to demonstrating a C run-time library call, this example shows some useful calling macros.

```
/* Non-Leaf Assembly Routines Example – _rms */
/* float rms(float x, float y); An assembly language subroutine
to return the rms z = (x^2 + y^2)^(1/2) */

#include <asm_sprt.h>

.section/pm seg_pmco;
.extern _sqrtf;
.global _rms;
_rms:
   entry;              /* first line of non-leaf routine */

   f4 = f4 * f4;
   f8 = f8 * f8;
   f4 = f4 + f8;
/* f4 contains argument to be passed to sqrtf function */

   ccall (_sqrtf);
/* use the ccall() macro to make a function call in a C
environment; f0 contains the result returned by the _sqrtf
function. In turn, _rms returns the result to its caller in f0
(and it is already there) */
   exit;               /* last line of non-leaf routine */
```

If a called function takes more than three single word parameters, the remaining parameters must be pushed on to the stack and popped off the stack after the function call. The following function could call the `_add5` routine shown in "Using the Stack for Arguments and Return (Add 5)" on page 1-232. Note that the last parameter must be pushed on the stack first.

```
/* Non-Leaf Assembly Routines Example — _calladd5 */
/* int calladd5 (void); An assembly language subroutine that
calls another routine with more than 3 parameters.
This example adds the numbers 1, 2, 3, 4, and 5. */

#include <asm_sprt.h>
.section/pm seg_pmco;
.extern _add5;
.extern _sum_of_5;
.global _calladd5;
_calladd5:

entry;
   r4 = 5;
/* the fifth parameter is stored in r4 for pushing onto stack  */
   puts=r4;              /* put fifth parameter in stack       */
   r4 = 4;
/* the fourth parameter is stored in r4 for pushing onto stack */
   puts=r4;              /* put fourth parameter in stack      */
   r4 = 1;               /* the first parameter is sent in r4  */
   r8 = 2;               /* the second parameter is sent in r8 */
   r12 = 3;              /* the third parameter is sent in r12 */

   ccall (_add5);
/* use the ccall macro to make a function call in a C environment */
   alter(2);
/* call the alter() macro to remove the two arguments from
the stack */
   r0 = dm(_sum_of_5);
/* _sum_of_5 is where add5 stored its result */
   exit;
```

Using Call Preserved Registers (Pass Array)

Some functions need to make use of registers that the run-time environment defines as *call preserved registers*. These registers, whose contents are preserved across function calls, are useful for variables whose lifetime spans a function call. The following example performs an operation on the elements of a C array using call preserved registers.

```
/* Non-Leaf Assembly Routines Example – _pass_array */
/* void pass_array(
float function(float),
float *array,
int length);
An assembly language routine that operates on a C array */

#include <asm_sprt.h>
.section/pm seg_pmco;
.global _pass_array;
_pass_array:
      entry;
      puts = i8;
/* This function uses a call preserved register, i8, because
it could be used by multiple functions, and this way it does
not have to be stored for every function call */

   r0 = i1;
   puts = r0;      /* i1 is also call preserved */

i8 = r4;
/* read the first argument, the address of the function to call */

i1 = r8;
/* read the second argument, the C array containing the data
to be processed */

r0 = r12;
/* read third argument, the number of data points in the array */

lcntr=r0, do pass_array_loop until lce;
/* loop through data points */
```

```
f4=dm(i1,m5);
/* get data point from array, store it in f4 as a parameter for
the function call */

ccall(m13,i8);        /* call the function */
pass_array_loop:
   dm(i1,m6)=f0;
/* store the return value back in the array */
   i1 = gets(1);      /* restore the value of i1 */
   i8 = gets(2);      /* restore the value of i8 */

exit;
```

Compiler C++ Template Support

The compiler provides template support C++ templates as defined in the ISO/IEC 14882:1998 C++ standard, with the exception that the export keyword is not supported.

Template Instantiation

Templates are instantiated automatically during compilation using a linker feedback mechanism. This involves compiling files, determining any required template instantiations, and then recompiling those files making the appropriate instantiations. The process repeats until all required instantiations have been made. Multiple recompilations may be required in the case when a template instantiation is made that requires another template instantiation to be made.

By default, the compiler uses a method called *implicit instantiation*, which is common practice, and results in having both the specification and definition available at point of instantiation. This involves placing template specifications in a header (for example, "`.h`") file and the definitions in a source (for example, "`.cpp`") file. Any file being compiled that includes a header file containing template specifications will instruct the compiler to implicitly include the corresponding "`.cpp`" file containing the definitions of the compiler.

For example, you may have the header file "`tp.h`"

```
template <typename A> void func(A var)
```

and source file "`tp.cpp`"

```
template <typename A> void func(A var)
{
...code...
}
```

Two files "file1.cpp" and "file2.cpp" that include "tp.h" will have file "tp.cpp" included implicitly to make the template definitions available to the compilation.

If there is a desire not to use the implicit inclusion method then the switch -no-implicit-inclusion should be passed to the compiler. In the example, we have been discussing, "tp.cpp" will then be treated as a normal source file and should be explicitly linked into the final product.

Regardless of whether implicit instantiation is used or not, the compilation process involves compiling one or more source files and generating a ".ti" file corresponding to the source files being compiled. These ".ti" files are then used by the prelinker to determine the templates to be instantiated. The prelinker creates a ".ii" file and recompiles one or more of the files instantiating the required templates.

The prelinker ensures that only one instantiation of a particular template is generated across all objects. For example, the prelinker ensures that if both "file1.cpp" and "file2.cpp" invoked the template function with an int, that the resulting instantiation would be generated in just one of the objects.

Identifying Uninstantiated Templates

If for some reason the prelinker is unable to instantiate all the templates that are required for a particular link then a link error will occur. For example:

```
[Error li1021] The following symbols referenced in processor 'P0'
could not be resolved:
    'Complex<T1> Complex<T1>::_conjugate() const [with T1=short]
[_conjugate__16Complex__tm__2_sCFv_18Complex__tm__4_Z1Z]' refer-
enced from '.\Debug\main.doj'
    'T1 *Buffer<T1>::_getAddress() const [with T1=Complex<short>]
[_getAddress__33Buffer__tm__19_16Complex__tm__2_sCFv_PZ1Z]'
referenced from '.\Debug\main.doj'
    'T1 Complex<T1>::_getReal() const [with T1=short]
```

```
[_getReal__16Complex__tm__2_sCFv_Z1Z]' referenced from
'.\Debug\main.doj'

Linker finished with 1 error
```

Careful examination of the linker errors reveals which instantiations have not been made. Below are some examples.

```
Missing instantiation:
   Complex<short> Complex<short>::conjugate()
Linker Text:
   'Complex<T1> Complex<T1>::_conjugate() const [with T1=short]
   [_conjugate__16Complex__tm__2_sCFv_18Complex__tm__4_Z1Z]'
   referenced from '.\Debug\main.doj'

Missing instantiation:
   Complex<short> *Buffer<Complex<short>>::getAddress()
Linker Text:
   'T1 *Buffer<T1>::_getAddress() const [with T1=Complex<short>]
   [_getAddress__33Buffer__tm__19_16Complex__tm__2_sCFv_PZ1Z]'
   referenced from '.\Debug\main.doj'

Missing instantiation:
   Short Complex<short>::getReal()
Linker Text:
   'T1 Complex<T1>::_getReal() const [with T1=short]
   [_getReal__16Complex__tm__2_sCFv_Z1Z]' referenced from
   '.\Debug\main.doj'
```

There could be many reasons for the prelinker being unable to instantiate these templates, but the most common is that the `.ti` and `.ii` files associated with an object file have been removed. Only source files that can contain instantiated templates will have associated `.ti` and `.ii` files, and without this information, the prelinker may not be able to complete its task. Removing the object file and recompiling will normally fix this problem.

Another possible reason for uninstantiated templates at link time is when implicit inclusion (described above) is disabled but the source code has been written to require it. Explicitly compiling the `.cpp` files that would normally have been implicitly included and adding them to the final link is normally all that is needed to fix this.

Another likely reason for seeing the linker errors above is invoking the linker directly. It is the compiler's responsibility to instantiate C++ templates, and this is done automatically if the final link is performed via the compiler driver. The linker itself contains no support for instantiating templates.

2 ACHIEVING OPTIMAL PERFORMANCE FROM C/C++ SOURCE CODE

This chapter provides guidance to help 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
- "Improving Conditional Code" on page 2-21
- "Loop Guidelines" on page 2-22
- "Using Built-In Functions in Code Optimization" on page 2-32
- "Smaller Applications: Optimizing for Code Size" on page 2-36
- "Using Pragmas for Optimization" on page 2-38
- "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 this chapter 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.

Choose the language as appropriate.

As the 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 has very similar performance to pure C source. Many higher-level C++ features (for example, those resolved at compilation, 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 include virtual functions or classes used to implement basic data types.

This section contains:

- "How the Compiler Can Help" on page 2-4
- "Data Types" on page 2-8
- "Getting the Most From IPA" on page 2-10

- "Indexed Arrays Versus Pointers" on page 2-15
- "Function Inlining" on page 2-16
- "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, statistical profiler, 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 1-40) to the compiler invocation. A "release" build from within VisualDSP++ automatically enables optimization.

Using the Statistical Profiler

Tuning an application begins with an understanding of which areas of the application are most frequently executed and therefore where improvements would provide the largest gains. The statistical profiling feature provided in VisualDSP++ is an excellent means for finding these areas. More details about how to use it may be found in the *VisualDSP++ 4.0 User's Guide*.

The particular advantage of statistical profiling is that it is completely unobtrusive. Other forms of profiling insert instrumentation into the code, disturbing the original optimization, code size and register allocation to some degree.

The best methodology is usually to compile with both optimization and debug information generation enabled. In this way, you can obtain a profile of the optimized code while retaining function names and line number information. This gives you accurate results that correspond directly to the C/C++ source. Note that the compiler optimizer may have moved code between lines.

You can obtain a more accurate view of your application if you build it optimized but without debug information generation. You can then obtain statistics that relate directly to the assembly code. The only problem with doing this may be in relating assembly lines to the original source. Do not strip out function names when linking, since keeping function names means you can scroll through the assembly window to instructions of interest.

In very complicated code, you can locate the exact source lines by counting the loops, unless they are unrolled. Looking at the line numbers in the assembly file (use the `-save-temps` switch to retain compiler generated assembly files, which have the `.s` filename extension) may also help. The compiler optimizer may have moved code around so that it does not appear in the same order as in your original source.

Using Profile-Guided Optimization

Profile-Guided Optimization (PGO) is an excellent way to tune the compiler's optimization strategy for the typical run-time behavior of a program. There are many program characteristics that cannot be known statically at compile-time but can be provided 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 1-44) to create an executable file containing the necessary instrumentation for gathering profile data. Optimization should also be enabled—for best results, use the `-O` (on page 1-40) 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 `-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 of the **Project Options** dialog box in VisualDSP++) 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 2-36).

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 becomes invalid. The compiler does not use profile data when it can detect that it is inaccurate. However, it is possible to change source code in a way that it is not 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 "Optimization Control" on page 1-64.

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 is 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 are not seen if you compile with the `-S` switch (on page 1-47). To see the assembly file when IPA is enabled, use the `-save-temps` switch (on page 1-48), and look at the `.s` file produced after your program has been built.

As an alternative to IPA, you can achieve many of the same benefits by adding pragma directives and other declarations such as `__builtin_aligned` to provide information to the compiler about how each function interacts with the rest of the program.

These directives are further described in "Using __builtin_aligned" on page 2-12 and "Using Pragmas for Optimization" on page 2-38.

Data Types

The compiler directly supports the following scalar data types.

Single-Word Fixed-Point Data Types: Native Arithmetic	
`char`	32-bit signed integer
`unsigned char`	32-bit unsigned integer
`short`	32-bit signed integer
`unsigned short`	32-bit unsigned integer
`int`	32-bit signed integer
`unsigned int`	32-bit unsigned integer
`long`	32-bit signed integer
`unsigned long`	32-bit unsigned integer
Floating-Point Data Types: Native Arithmetic	
`double`	32-bit float-point
`float`	32-bit float-point
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 2-32.

Avoiding Emulated Arithmetic

Arithmetic operations for some types are implemented by library functions because the processor 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 2-8.

The hardware does not provide direct support for division, so division and modulus operations are almost always multicycle operations, even on integral type inputs. If the compiler has to issue a full-division operation, it usually requires to generate a call to a library function. One notable situation in which a library call is avoided is for integer division when the divisor is a compile-time constant and is a power of two. In that case, the compiler generates a shift instruction. Even then, a few fix-up instructions are needed after the shift if the types are signed. If you have a signed division by a power of two, consider whether you can change it to unsigned in order to obtain a single-instruction 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 suffers not only because the operation takes multiple cycles, but also because the effectiveness of the compiler optimizer is 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 has 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.

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 Statically

IPA identifies variables that have only one value and replaces them with constants, resulting in a host of benefits for the optimizer's analysis. For this to happen a variable must have a single value throughout the program. If the variable is statically initialized to zero, as 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 does 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() {
   printf("val %d",val);
}
int main() {
   init();
   func();
}
```

Bad: IPA cannot see that `val` is a constant

The code 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.

Dual Word-Aligning Your Data

This and the following section applies to the dual compute-block architectures used with the ADSP-2116x, ADSP-2126x and ADSP-2136x processors.

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.

For external data, the ADSP-2116x chips require that dual-word memory accesses reference dual-word-aligned addresses. Therefore, for the most efficient code to be generated, you should ensure that data buffers are dual-word-aligned.

The compiler helps to establish the alignment of array data. Top-level arrays are allocated at dual-word-aligned addresses, regardless of their data types. To do this for local arrays means that the compiler also ensures that stack frames are kept dual-word-aligned. 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 through these input arrays one element at a time, starting at element zero), then IPA should be able to establish that the alignment is suitable for full-width accesses.

Where an inner loop processes a single row of a multi-dimensional array, try to ensure that each row begins on a word boundary. In particular, two-dimensional arrays should be defined in a single block of memory rather than as an array of pointers to rows all separately allocated with `malloc`. It is difficult for the compiler to keep track of the alignment of the pointers in the latter case. It may also be necessary to insert dummy data at the end of each row to make the row length a multiple of two words.

Using __builtin_aligned

To avoid the need to use IPA to propagate alignment, and for situations when IPA cannot guarantee the alignment but you can, 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. It is important to remember when you add this declaration that you are responsible for making sure that it is valid, and that, if the assertion is not true, then the code produced by the compiler is likely to malfunction.

The assertion is particularly useful for function parameters, although you may assert that any pointer is aligned. 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 dual-word boundaries. To assert instead that both pointers are always aligned one `char` past a dual-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. The second parameter should give the alignment in words as a literal constant.

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) {
int i;
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 IPA is enabled, the compiler looks 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.

```
fn(glob1, glob2, N);
fn(glob3, glob1, N);
```

Bad: sets intersect - both `a` and `b` may access `glob1`; `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 Versus Pointers

The C language allows a program to access data from an array in two ways: either by indexing from an invariant base pointer, or by incrementing a pointer. The following two versions of vector addition illustrate the two styles:

Style 1: using indexed arrays

```
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 generates better optimized code than the other, but it is not always the same style that is better.

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—sometimes this is accomplished better by hand.

The best strategy is to start with array notation. If the generated code looks unsatisfactory, try using pointers. Outside the critical loops, use the indexed style, since it is easier to understand.

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 1-41). This switch automatically enables optimization.

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 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 automatically inlines small functions where possible. If the application has a tight upper code-size limit, the resulting code-size expansion may be too great. It is worth considering using automatic inlining in conjunction with the `-Ov num` switch (on page 1-41) to restrict inlining (and other optimizations with a code-size cost) to parts of the application that are performance-critical. This is 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 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 functions 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 1-102.

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 are 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 2-32.

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.

Try to put arrays into different memory sections.

The processor hardware can support two memory operations on a single instruction line, combined with a compute instruction. However, two memory operations complete in one cycle only if the two addresses are situated in different memory blocks—if both access the same block, then a stall is incurred.

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 "Dual Memory Support Keywords (pm dm)" on page 1-90). 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 the SHARC processor is in one unified address space (except for the ADSP-21020 processor) and there is no restriction concerning in which part of memory program code or data can be placed. However, the default `.LDF` files ensure that `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 memory block used for `pm`-qualified data in the default `.LDF` files is the same memory block as is used for the program code, hence the name "Program Memory".

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;
```

to allow simultaneous accesses to the two buffers.

Note that the explicit placement of data in Program Memory can only be done for global or static data.

Improving Conditional Code

When compiling conditional statements, the compiler attempts to predict whether the condition will usually evaluate to true or to false, and will arrange for the most efficient path of execution to be that which is expected to be most commonly executed.

You can use the `expected_true` and `expected_false` built-in functions to control the compiler's behavior for specific cases. By using these functions, you can tell the compiler which way a condition is most likely to evaluate, and so influence the default flow of execution. For example,

```
if (buffer_valid(data_buffer))
   if (send_msg(data_buffer))
      system_failure();
```

shows two nested conditional statements. If it was known that, for this example, `buffer_valid()` would usually return true, but that `send_msg()` would rarely do so, the code could be written as

```
if (expected_true(buffer_valid(data_buffer)))
   if (expected_false(send_msg(data_buffer)))
      system_failure();
```

See "Compiler Performance Built-in Functions" on page 1-108 (on `expected_true` and `expected_false` functions) for more information.

The compiler can also determine the most commonly-executed branches automatically, using Profile-Guided Optimization. See "Optimization Control" on page 1-64 for more details.

Loop Guidelines

Loops are where an application ordinarily spends the majority of its time. It is therefore useful to look in detail at how to help the compiler to produce the most efficient code.

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 causes a decrease in code density and execution performance.

Avoiding Unrolling Loops

Do not unroll loops yourself.

Not only does loop unrolling make the program harder to read but 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 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];
```

```
        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 1-38).

Avoiding Loop Rotation by Hand

Do not rotate loops by hand.

Programmers are often tempted to "rotate" loops in processor 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.

For example,

```
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: is 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++) {
       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 is 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

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 runs 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 is able to convert `IF-THEN-ELSE` and `?:` constructs into conditional instructions. In other cases, it is 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 usually succeeds in transforming conditional code equivalent to `min` or `max` into the single instruction. With particularly convoluted code the transformation may be missed, in which case it is better to use `min` or `max` in the source code.

The compiler does 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)
      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 usually is unable to generate a hardware loop if the loop contains a function call due to the expense of saving and restoring the context of a hardware loop. In addition 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 2-8.

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 needs to work out the loop trip count. To do so, it must divide `n` by 3. The compiler decides that this is worthwhile as it speeds 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.

Loop Control

Use `int` types for loop control variables and array indices.

Use automatic variables for loop control and loop exit test.

For loop control variables and array indices, it is always better to use signed `int`s 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. The `restrict` keyword currently has no effect when used on variables defined in a scope nested within the outermost scope of a function.

Using the Const Qualifier

By default, the compiler assumes that the data referenced by a pointer to `const` type do 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.

The use of `const` in the above example has a similar effect on the `no_alias` pragma (see "#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 generates 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`.

Avoiding Long Latencies

All pipelined machines introduce stall cycles when you cannot execute the current instruction until a prior instruction has exited the pipeline. For example, the SHARC processor stalls for three cycles on a table lookup; `a[b[i]]` takes three 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++ 4.0 IDDE. By single-stepping through the program, you can 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 processor 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 1-102), it presents some code examples where implementation choices are available to the programmer.

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 the built-in functions also usually results in improved code readability.

For more information on supported built-in functions, refer to "Compiler Built-In Functions" on page 1-102.

Examples of the two styles are:

```
asm("#include <def21060.h>");  // Bit definitions for the registers

void func_no_interrupts(void){
   // Check if interrupts are enabled.
   // If so, disable them, call the function, then re-enable.
   int enabled;
   asm("r0=0; bit tst MODE1 IRPTEN; if tf r0=r0+1; %0 = r0;"
      : "=d"(enabled) : : "r0");
   if (enabled)
```

```
      asm("bit clr mode1 IRPTEN;");      // Disable interrupts
   func();                               // Do something
   if (enabled)
      asm("bit set mode1 IRPTEN;");      // Re-enable interrupts
}
```

Bad: uses inline `asm` statement.

```
#include <sysreg.h>         // Sysreg functions
#include <def21060.h>       // Bit definitions for the registers

void func_no_interrupts(void){
   // Check if interrupts are enabled.
   // If so, disable them, call the function, then re-enable.
   int enabled = sysreg_bit_tst(sysreg_MODE1, IRPTEN);
   if (enabled)
      sysreg_bit_clr(sysreg_MODE1, IRPTEN);  // Disable interrupts
   func();                                   // Do something
   if (enabled)
      sysreg_bit_set(sysreg_MODE1, IRPTEN);
                                             // Re-enable interrupts
}
```

Good: uses `sysreg.h`.

This example reads and returns the `CYCLES` register.

Using Circular Buffers

Circular buffers are often extremely useful in processor 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 produces a hardware circular buffer instruction for this access.

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.

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 1-102).

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 depends 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 1-41), which may be used in conjunction with IPA, performs every performance-enhancing transformation except those that increase code size. In addition, the `-e` linker switch (`-flags-link -e` if used from the compiler command line) may be helpful (on page 1-29). This operation 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 is 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 produces significantly slower code than optimizing for speed.

The compiler provides a way to balance between the two extremes of `-O` and `-Os`. This is the sliding-scale `-Ov num` switch (adjustable using the optimization slider bar under **Project Options** in the VisualDSP++ IDDE), described on page 1-41. 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 is used 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 1-64) 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.

Using Pragmas for Optimization

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 1-110 for full details of how each pragma works; the emphasis here is 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 normally are ignored by a compiler that does not recognize them.

Function Pragmas

Function pragmas include `#pragma alloc`, `#pragma const`, `#pragma pure`, `#pragma result_alignment`, `#pragma regs_clobbered`, and `#pragma optimize_{off|for_speed|for_space|as_cmd_line}`,

#pragma alloc

This pragma asserts that the function behaves like the `malloc` library function. In particular, it returns a pointer to new memory that cannot alias any pre-existing buffers. In the following code,

```
#pragma alloc
int *new_buf(void);
int *vmul(int *a, int *b) {
   int i;
```

```
    int *out = new_buf();
    for (i=0; i<100; i++)
       out[i] = a[i] * b[i];
}
```

Good: uses `#pragma alloc` to disambiguate out from `a` and `b`.

the use of the pragma allows the compiler to be sure that the write into buffer `out` does not modify either of the two input buffers `a` or `b`, and, therefore, the iterations of the loop may be re-ordered.

#pragma const

This 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 always yield the same result. In this way, calls to `#pragma const` functions may be hoisted out of loops if their parameters are loop independent.

#pragma pure

Like `#pragma const`, this pragma asserts to the compiler that a function does not have any side effects (such as modifying global variables or data buffers). However, the result returned may be a function of both the parameter values and any global variables. The pragma may be applied to a function prototype or definition. Two calls to the function with identical parameters 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 result_alignment

This pragma may be used on functions that have either pointer or integer results. When a function returns a pointer, the pragma is used to assert that the return result always has some specified alignment. Therefore, the example might be further refined if it is known that the `new_buf` function always returns buffers which are aligned on a dual-word boundary.

```
#pragma alloc
#pragma result_alignment (2)
int *new_buf(void);

int *vmul(int *a, int *b) {
   int i;
   int *out = new_buf();
   for (i=0; i<100; i++)
      out[i] = a[i] * b[i];
}
```

Good: uses pragma `result_alignment` to specify that out has strict alignment.

Further details on this pragma may be found in "#pragma result_alignment (n)" on page 1-127. Another more laborious way to achieve the same effect would be to use `__builtin_aligned` at every call site to assert the alignment of the returned result.

#pragma regs_clobbered

This 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:
   modify(i7,-3);
   r2=255;
   r8=r8+r4;
   r0=r8 and r2;
   i12=dm(m7,i6);;
   jump(m14,i12)(DB); rframe; nop;
._add_mask.end
```

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 following prototype,

```
#pragma regs_clobbered "r0, r2, i12, ASTAT"
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 does not need many registers when compiled, it can be defined using:

```
#pragma regs_clobbered "r0, r2, i12, 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 `r0, r2, i12` and the condition codes are not modified by the function. If any other registers are used in the compilation of the function, they are saved and restored during the function prologue and epilogue.

In general, it is not 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 `ASTAT` in the clobbered set above. For more information, refer to "#pragma regs_clobbered string" on page 1-123.

#pragma optimize_{off|for_speed|for_space|as_cmd_line}

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 `cc21k` command line when the compiler was invoked. Refer to "General Optimization Pragmas" on page 1-119 for more information.

Loop Optimization Pragmas

Many pragmas are targeted towards helping to produce optimal code for inner loops. These are the `loop_count`, `no_vectorization`, `vector_for`, `all_aligned`, 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 1-117.

#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 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 checks various properties of the loop and does not vectorize it if it believes that it is unsafe or it is not possible to 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 1-118.

#pragma SIMD_for

The `#pragma SIMD_for` is similar to the `vector_for` pragma but makes the weaker assertion that only two iterations may be issued in parallel. Further details are given in section (see "#pragma SIMD_for" on page 1-116).

#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 word boundary at the beginning of the first iteration.

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, 4);
__builtin_aligned(b, 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 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.

#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 1-118 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
`-const-read-write` on page 1-26	Specifies that data accessed via a pointer to `const` data may be modified elsewhere.
`-flags-link -e` on page 1-29	Specifies linker section elimination.
`-force-circbuf` on page 1-30	Treats array references of the form `array[i%n]` as circular buffer operations.
`-ipa` on page 1-33	Turns on inter-procedural optimization. Implies use of `-O`. May be used in conjunction with `-Os` or `-Ov`.
`-no-fp-associative` on page 1-38	Does not treat floating-point multiply and addition as an associative.
`-no-saturation` on page 1-39	Do not turn non-saturating operations into saturating ones.
`-O` on page 1-40	Enables code optimizations and optimizes the file for speed.
`-Os` on page 1-41	Optimizes the file for size.
`-Ov num` on page 1-41	Controls speed vs. size optimizations (sliding scale).
`-pguide` on page 1-44	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization.
`-save-temps` on page 1-48	Saves intermediate files (for example, `.s`).

Example: How the Optimizer Works

The following floating-point scalar product loop are used to show how the compiler optimizer works.

Example: C source code for floating-point scalar product.

```
float sp(float *a, float *b, int n) {
   int i;
   float sum=0;
   __builtin_aligned(a, 2);
   __builtin_aligned(b, 2);
   for (i=0; i<n; i++) {
      sum+=a[i]*b[i];
   }
   return sum;
}
```

After code generation and conventional scalar optimizations, the compiler generates a loop that looks something like the following example.

Example: Initial code generated for floating-point scalar product.

```
   lcntr = r3, do(pc, .P1L10-1)until lce;
.P1L9:
   r4 = dm(i1, m6);
   r2 = dm(i0, m6);
   f12 = f2 * f4;
   f10 = f10 + f12;
   // end_loop .P1L9;
.P1L10:
```

The loop exit test has been moved to the bottom and the loop counter rewritten to count down to zero. This enables a zero-overhead hardware loop to be created (`r3` is initialized with the loop count). `sum` is being accumulated in `r10`. `i0` and `i1` hold pointers that are initialized with the parameters `a` and `b` and incremented on each iteration.

The ADSP-2116x, ADSP-2126x and ADSP-2136x processors have two compute units that may perform computations simultaneously. In order to use both these compute blocks, the optimizer unrolls the loop to run two iterations in parallel. `sum` is now being accumulated in `r10` and `s10`, which must be added together after the loop to produce the final result. In order to use the dual-word loads needed for the loop to be as efficient as this, the compiler has to know that `i0` and `i1` have initial values that are even. This is done in the above example by use of `__builtin_aligned`, although it could also be propagated with IPA.

Note also that unless the compiler knows that original loop was executed an even number of time, a conditionally-executed odd iteration must be inserted outside the loop. `r3` is now initialized with half the value of the original loop.

Example: Code generated for floating-point scalar product after vectorization transformation.

```
   bit set mode1 0x200000; nop;      // enter SIMD mode
   m4 = 2;
   lcntr = r3, do(pc, .P1L10-1)until lce;
.P1L9:
   r4 = dm(i1, m4);
   r2 = dm(i0, m4);
   f12 = f2 * f4;
   f10 = f10 + f12;
   // end_loop .P1L9;
.P1L10:
   bit clr mode1 0x200000; nop;      // exit SIMD mode
```

Finally, the optimizer rotates the loop, unrolling and overlapping iterations to obtain highest possible use of functional units. Code similar to the following is generated if it were known that the loop was executed at least four times and the loop count was a multiple of two.

Example: Code generated for floating-point scalar product after software pipelining.

```
   bit set mode1 0x200000; nop;      // enter SIMD mode
   m4 = 2;
   r4 = dm(i1, m4);
   r2 = dm(i0, m4);
   lcntr = r3, do(pc, .P1L10-1)until lce;
.P1L9:
   f12 = f2 * f4, r4 = dm(i1, m4);
   f10 = f10 + f12, r2 = dm(i0, m4);
   // end_loop .P1L9;
.P1L10:
   f12 = f2 * f4;
   f10 = f10 + f12;
   bit clr mode1 0x200000; nop;      // exit SIMD mode
```

If the original source code is amended to declare one of the pointers with the pm qualifier, the following optimal code is produced for the loop kernel.

Example: Code generated for floating-point scalar product when one buffer placed in PM.

```
   bit set mode1 0x200000; nop;      // enter SIMD mode
   m4 = 2;
   r5 = pm(i1, m4);
   r2 = dm(i0, m4);
   r4 = pm(i1, m4);
   f12 = f2 * f5, r2 = dm(i0, m4);
   lcntr = r3, do(pc, .P1L10-1)until lce;
.P1L9:
   f12 = f2 * f4, f10 = f10 + f12, r2 = dm(i0, m4), r4 = pm(i1, m4);
   // end_loop .P1L9;
.P1L10:
   f12 = f2 * f4, f10 = f10 + f12;
   f10 = f10 + f12;
   bit clr mode1 0x200000; nop;      // exit SIMD mode
```

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 1-47), the `-save-temps` switch (see on page 1-48) 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
- “Vectorization” on page 2-60
- “Modulo Scheduling” on page 2-65

Procedure Statistics

For each function call, the following is reported:

- Frame size: size of stack frame.
- Registers used. Since function calls tend to implicitly clobber registers, there are several sets:

    1. The first set is composed of the scratch registers changed by the current function, 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 1

Consider the following program:

```
struct str {
    int x1, x2;
};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:
//
//   Frame size             = 6 words
//
// Scratch registers modified:
             {r0,r2,r4,r8,r12,i12,Bi6-Bi7,Bi12,acc}
//
// Call preserved registers used:{r3,i0}
//
// Registers clobbered by function calls:
   {r0-r2,r4,r8,r12,i4,i12-i13,b4,b12-b13,m4,m12,s0-s15,
    Bi4,Bi12-Bi13,Bm4,Bm12,ustat1-ustat4,acc,mcc,scc,
    btf,lcntr,smrf,smrb,sacc,smcc,sscc,sbtf}
//------------------------------------------------------------
// line "example1.c":6
      modify(i7,-7);
      r2=i0;
      dm(-8,i6)=r3;
      dm(-7,i6)=r2;
// line 11
      r12=-6;
      r4=r12+1,
      /*||*/ r2=i6;
      r8=r2+r12,
      /*||*/ r2=i6;
      r4=r2+r4,
      /*||*/ i0=i6;
// line 12
//   -- bubble --
      modify(i0,-4);
// line 11
      cjump _func1 (DB); dm(i7,m7)=r2; dm(i7,m7)=pc;
      r3=pass r0,
      /*||*/ r2=dm(i0,m7);
// line 12
      dm(i7,m7)=r2;
```

```
        r2=dm(i0,m7);
        dm(i7,m7)=r2;
        cjump _func2 (DB); dm(i7,m7)=r2; dm(i7,m7)=pc;
        r0=r3+r0;
        modify(i7,2);
        r3=dm(-8,i6);
        i12=dm(m7,i6);
        jump(m14,i12) (DB);
        i0=dm(-7,i6);
//    -- bubble --
        rframe;

_foo.end:
   .global _foo;
   .type _foo,STT_FUNC;
```

Notes:

- The set of Scratch registers modified is `{r0,r2,r4,r8,r12,i12,Bi6-Bi7,Bi12,acc}` 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 `{r3, i0}` 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 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:

- 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.

- 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 2 (for ADSP-21060 Processor)

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 .....................
//   Frame size            = 2 words
//
// Scratch registers modified:{r0,r2,i12,Bi6-Bi7,Bi12,acc,lcntr}
//
// No call preserved registers used.
```

```
//------------------------------------------------------------
// Original Loop at "example2.c" line 6 col 5 --  loop structure
// removed due to constant propagation.
//------------------------------------------------------------
// Original Loop at "exampleC.c" line 6 col 5 -- loop structure
   removed due to constant propagation.
//------------------------------------------------------------
// line "example2.c":1
      modify(i7,-2);
      r0=,0;
// line 4
      lcntr=9999, do(pc,.P1L2-1)until lce;

.P1L1:
//------------------------------------------------------------
// Loop at "example2.c" line 4 col 5
//------------------------------------------------------------
// This loop executes 1 iteration of the original loop in 2 cycles.
//------------------------------------------------------------
// This loop's resource usage is:
//------------------------------------------------------------
// line 5
      r2=r0+1;
      r0=r0+r2;
// line 4
//    end loop .P1L1:
//------------------------------------------------------------
// End Loop L1
//------------------------------------------------------------

.P1L2:
//------------------------------------------------------------
// Part of top level (no loop)
//------------------------------------------------------------
      i12=dm(m7,i6);
      jump(m14,i12)(DB); rframe; nop;

_bar.end:
   .global _foo;
   .type _foo,STT_FUNC;
```

Notes:

- The keywords identifying the two loops are:
  1. `for` — located at line 4, column 5
  2. `while` — located at line 6, column 5
- Immediately after the procedure statistics, there is a message stating that the loop at line 6 in the user program was removed. The compiler recognized that the value of i after the first loop is 9999 and that the second loop is not executed.
- The start of the loop at line 4 is marked in the assembly by the `'Loop at "example2.c" line 4 col 5'` annotation. This annotation follows the loop label `.P1L1` which is used to identify the end of the loop "`End Loop .P1L1`".

File Position

As seen in the Example 2 (in "Loop Identification Annotations" on page 2-55), a file position is given using the file name, line number and the column number in that file as `"example2.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 3 (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 "example3.c" line 16 col 5 inlined from
   "example2.3.c" line 23 col 7
//--------------------------------------------------------------
// "example3.c" line 8 col 5
    ...
    ...

.P2L4:
//--------------------------------------------------------------
```

```
// Loop at "example3.c" line 8 col 5 inlined from
   "example2.3.c" line 17 col 4 inlined from "example2.3.c"
   line 23 col 7
//--------------------------------------------------------------
...
//--------------------------------------------------------------
// End Loop L4
//--------------------------------------------------------------

...

.P2L9:
//--------------------------------------------------------------
// Loop at "example2.3.c" line 8 col 5 inlined from "example2.3.c"
   line 18 col 4 inlined from "example2.3.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 goes around.

Under certain conditions, the compiler is able to take two operations executed in consecutive iterations of a loop and to execute them in a single more powerful SIMD instruction giving a loop with a smaller trip count. The transformation in which operations from two subsequent iterations are executed in one SIMD 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 250. The vectorization factor is the number of operations in the original loop that are executed at once in the transformed loop. It is illustrated using some pseudo code below.

```
for(i=0; i< 1000; i+=2)
   (a[i], a[i+1]) = (b[i],b[i+1]) .plus2. (c[i], c[i+1]);
```

In the above example, the vectorization factor is 2. A loop may be vectorized more than once.

If the trip count is not a multiple of the vectorization factor, some iterations need to be peeled off and executed unvectorized. 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< 1000; i+=2)
   (a[i], a[i+1]) = (b[i],b[i+1]) .plus2. (c[i], c[i+1]);
   a[1000] = b[1000] + c[1000];
             // This is one iteration peeled from
             // the back of the loop.
```

In the above examples the trip count is known and the amount of peeling is also known. If the trip count 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.

Loop Flattening

Another transformation, slightly related to vectorization, is loop flattening. The loop flattening operation takes two nested loops that run `N1` and `N2` times, respectively, and transforms them into a single loop that runs `N1*N2` times. For instance, the following function

```
void copy_v(int a[][100], int b[][100]) {
   int i,j;
   for (i=0; i< 30; ++i)
#pragma SIMD_for
```

```
#pragma no_alias
   for (j=0; j < 100; ++j)
       a[i][j] = b[i][j];
}
```

is transformed into

```
void copy_v(int a[][100], int b[][100]) {
   int i,j;
   int *p_a = &a[0][0];
   int *p_b = &b[0][0];
   for (i=0; i< 3000; ++i)
       p_a[i] = p_b[i];
}
```

This may further facilitate the vectorization process:

```
void copy_v(int a[][100], int b[][100]) {
   int i,j;
   int *p_a = &a[0][0];
   int *p_b = &b[0][0];
   for (i=0; i< 3000; i+=2)
       (p_a[i], p_a[i+1]) = (p_b[i], p_b[i+1]);
}
```

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 loop flattening, it is necessary to account for the loop that is lost and show the remaining loop that it was merged with.

Example 4 (Vectorization for ADSP-21160 Processor):

Consider the test program:

```
void add(int *a, int *b, int* c, int dim) {
   int i;
#pragma no_alias
#pragma SIMD_for
   for (i = 0 ; i < dim; ++i)
       a[i] = b[i] + c[i];
}
```

for which the vectorization information is:

```
.P1L7:
//------------------------------------------------------------------
// Loop at "vector.c" line 4 col 5
//------------------------------------------------------------------
// This loop executes 2 iterations of the original loop in 4 cycles.
//------------------------------------------------------------------
// This loop's resource usage is:
//------------------------------------------------------------------
// Loop was vectorized by a factor of 4.
//------------------------------------------------------------------
// Vectorization peeled 1 conditional iteration from the back of
// the loop because of an unknown trip count, possible not a
// multiple of 2.
//------------------------------------------------------------------
// line 5
      r2=dm(i4,m4);
      r12=dm(i1,m4);
      r2=r2+r12;
      dm(i0,2)=r2;
// line 4
// end loop .P1L7;
//------------------------------------------------------------------
// End Loop L7
//------------------------------------------------------------------
```

```
.P1L8:

.P1L12:
//------------------------------------------------------------------
// Part of top level (no loop)
//------------------------------------------------------------------
      bit clr mode1 0x200000; nop;
      r2=1;
      r12=r1 and r2;
      comp(r12, r2);
      if ne jump(pc,.P1L19);

.P1L4:
// line 5
      r2=dm(m5,i4);
      r12=dm(m5,i1);
      jump(pc,.P1L19) (DB);
      r2=r2+r12;
      dm(m5,i0)=r2;
```

In this example, the vectorization factor is 2. Since the trip count '`dim`' is unknown, one conditional iteration is peeled from the back of the loop, corresponding to the case where '`dim`' is `2k+1`.

Loop Flattening Example:

The assembly annotations for the `copy_v` function (from "Loop Flattening" on page 2-61) are:

```
_copy_v:
//------------------------------------------------------------
..................... Procedure statistics ...................
//------------------------------------------------------------ --
// Original Loop at "copy_y.c" line 6 col 5 --  loop flatten into
   loop at "copy_y.c" line 9 col 2
//------------------------------------------------------------
...................... procedure code .......................
.P1L1:
//------------------------------------------------------------
// Loop at "copy_y.c" line 9 col 2
```

```
//------------------------------------------------------------
...Loop annotations ...
...Loop body ...
   //end loop .P1L1;
//------------------------------------------------------------
// 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
- The maximum trip count, if it is known and the trip count is unknown
- The trip modulo, if it is known and the trip count is unknown
- The stage count (iterations in parallel)
- The MVE unroll factor
- The resource usage
- The Minimum initiation interval due to resources (`res MII`)

The compiler's `-annotate-loop-instr` switch (on page 1-25) can be used to produce additional 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 IDs 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.
- 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 *b, int *c) {
2      int i;
3   #pragma no_alias
4      for (i = 0; i < 200; ++i)
5          a[i] = b[i] + c[i];
6   }
```

The annotated output for the modulo scheduling, including the optional annotation of the modulo scheduling related instructions, is:

```
7  _add:
8   //-------------------------------------------------------------
9   //.................Procedure statistics.....................
10  //
11  //    Frame size            = 4 words
12  //
```

```
13  // Scratch registers
       modified:{r2,r12,i4,i12,Bi4,Bi6-Bi7,Bi12,acc,lcntr}
14  //
15  // Call preserved registers used:{i0-i1}
16
//-------------------------------------------------------------
17  // line "modulo.c":1
18        modify(i7,-6);
19        r2=i0;
20        dm(-7,i6)=r2;
21        r2=i1;
22        dm(-6,i6)=r2;
23  //        -- 2 bubbles --
24        i4=r8;
25        i0=r12;
26        i1=r4;
27  // line 7
28  //        -- bubble --
29        r2=dm(i4,m6);          // {L8 prolog:id=2,sn=0,rs  =0}
30        r12=dm(i0,m6);         // {L8 prolog:id=4,sn=0,rs  =0}
31        lcntr=199, do(pc,.P1L14-1)until lce;
32
33  .P1L8:
34
//-------------------------------------------------------------
35  // Loop at "modulo.c" line 6 col 5
36  //
37  // This loop executes 1 iteration of the original loop in 3 cycles.
38
//-------------------------------------------------------------
39  // Trip Count  =  199
40
//-------------------------------------------------------------
41  // Successfully found modulo schedule with:
42  //      Initiation Interval (II)       = 3
43  //      Stage Count (SC)               = 2
44  //      MVE Unroll Factor              = 1
45  //      Minimum initiation interval due to resources
            (res MII)  = 3.00
46
//-------------------------------------------------------------
47  // This loop's resource usage is:
```

```
48
//------------------------------------------------------------ -
49       r12=r12+r2,       // {L8 kernel:id=1,sn=1,rs=0,Iter=0}
50       /*||*/ r2=dm(i4,m6);
                           // {L8 kernel:id=2,sn=0,rs =0,Iter=1}
51       dm(i1,m6)=r12;    // {L8 kernel:id=3,sn=1,rs =0,Iter=0}
52       r12=dm(i0,m6);    // {L8 kernel:id=4,sn=0,rs =0,Iter=1}
53       // end loop .P1L8;
54
//-------------------------------------------------------------
55  // End Kernel for Loop L8
56
//-------------------------------------------------------------
57
58  .P1L14:
59
//-------------------------------------------------------------
60  // Part of top level (no loop)
61
//-------------------------------------------------------------
62       r2=r12+r2;         // {L8 epilog:id=1,sn=1,rs =0}
63       dm(i1,m6)=r2;      // {L8 epilog:id=3,sn=1,rs =0}
64       i0=dm(-7,i6);
65  //       -- bubble --
66       i12=dm(m7,i6);
67       jump(m14,i12) (DB);
68       i1=dm(-6,i6);
69  //       -- bubble --
70       rframe;
71
72  ._add.end:
```

Notes:
Lines 39-47 define the kernel information: loop name and modulo schedule parameters: II, stage count, etc.

Lines 49-53 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 50 looks like:

```
50   /*||*/ r2=dm(i4,m6);  // {L8 kernel:id=2,sn=0,rs =0,Iter=1}
```

This annotation describes:

- The fact that this instruction belongs to the kernel of the loop starting at `L8`.
- `sn=0` shows that this instruction belongs to stage count 0.
- `rs=0` shows that this instruction uses register set 0.
- `Iter=1` specifies that this instruction belongs to the second iteration of the original loop (`Iter` numbers are zero-based).

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:

```
29      r2=dm(i4,m6);     // {L8 prolog:id=2,sn=0,rs =0}
68      dm(i1,m6)=r2;     // {L8 epilog:id=3,sn=1,rs =0}
```

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:

```
63 r4 = pass r4,
        dm(i1,m6)=r2;     // {L8 epilog:id=3,sn=1,rs =0}
```

This shows a line with two instructions. The second instruction "`r4=pass r4`" 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. Many functions are implemented in the ADSP-21xxx assembly language. C and C++ programs depend on library functions to perform operations that are basic to the C and C++ programming environments. These operations include memory allocations, character and string conversions, and math calculations. Using the library simplifies your software development by providing code for a variety of common needs.

The sections of this chapter present the following information on the compiler:

- "C and C++ Run-Time Libraries Guide" on page 3-3
  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 `cc21k` compiler.
- "C Run-Time Library Reference" on page 3-64
  contains reference information about the C run-time library functions included with this release of the `cc21k` compiler.

The `cc21k` compiler provides a broad collection of library functions, including those required by the ANSI standard and additional functions supplied by Analog Devices that are of value in signal processing applications. In addition to the standard C library, this release of the compiler software includes the abridged C++ library, a conforming subset of the standard C++ library. The abridged C++ library includes the embedded C++ and embedded standard template libraries.

This chapter describes the standard C/C++ library functions that are supported in the current release of the run-time libraries. Chapter 4, "DSP Library for ADSP-2106x and ADSP-21020 Processors", and Chapter 5, "DSP Library for ADSP-2116x/2126x/2136x Processors", describe a number of signal processing, matrix, and statistical functions that assist 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 Function*s, Englewood Cliffs, New Jersey: Prentice Hall, 1980. For more information on the C++ library portion of the ANSI/ISO Standard for C++, see Plauger, P. J. (Preface), *The Draft Standard C++ Library,* Englewood Cliffs, New Jersey: Prentice Hall, 1994, (ISBN: 0131170031).

The C++ library reference information in HTML format is included on the software distribution CD-ROM. To access the reference files from VisualDSP++, use the **Help Topics** command (**Help** menu) and select the **Reference** book icon. From the **Online Manuals** topic, you can open any of the library files. You can also manually access the HTML files using a web browser.

C and C++ Run-Time Libraries Guide

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

- "Calling Library Functions" on page 3-3
- "Linking Library Functions" on page 3-4
- "Working with Library Header Files" on page 3-12
- "Calling Library Functions from an ISR" on page 3-29
- "Using Compiler Built-In C Library Functions" on page 3-30
- "Abridged C++ Library Support" on page 3-32
- "Measuring Cycle Counts" on page 3-40
- "File I/O Support" on page 3-49

For information on the C library's contents, see "C Run-Time Library Reference" (starting on on page 3-64). For information on the Abridged C++ library's contents, see "Abridged C++ Library Support" (starting on on page 3-32) 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 "C Run-Time Library Reference" (starting on on page 3-64) section 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 Header Files" on page 3-12.

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 Interface" on page 1-209.

You can use the archiver, `elfar`, described in the *VisualDSP++ 4.0 Linker and Utilities Manual*, to build library archive files of your own functions.

Linking Library Functions

The C/C++ run-time library is organized as four libraries:

- C run-time library — Comprises all the functions that are defined by the ANSI standard
- C++ run-time library
- DSP run-time library — Contains additional library functions supplied by Analog Devices that provide services commonly required by DSP applications
- I/O library — Supports a subset of the C standard's I/O functionality

In general, several versions of the C/C++ run-time library are supplied in binary form; for example, variants are available for different SHARC architectures and are listed in Table 3-1, Table 3-2, Table 3-3, and Table 3-4. Some versions of these binary files are also built for running in a multi-threaded environment; these binaries have `mt` in their filename.

In addition to regular run-time libraries, VisualDSP++ 4.0 compiler uses `libio*_lite.dlb` libraries which provide smaller versions of `LibIO` with more limited functionality. These smaller `LibIO` libraries can be used by

specifying the `-flags-link -MD__LIBIO_LITE` switch on the build command line. The VisualDSP++ 4.0 compiler also supports the C++ exception handling support library, `libeh*.dlb`.

Table 3-1 contains a list of the run-time libraries and start-up files that have been built for the ADSP-21020 and ADSP-2106x processors, and are installed in the subdirectory `21k\lib`.

Table 3-1. C and C++ Files and Libraries for ADSP-210xx Processors

Description	Library Name	Comments

C run-time library	`libc.dlb`
	`libc020.dlb`
	`libcmt.dlb`
		ADSP-21020 processor only

C++ run-time library g	`libcpp.dlb` `libcppmt.dlb`	
C++ run-time support library	`libcpprt.dlb` `libcpprtmt.dlb`	
C++ run-time library with exception handling	`libcppeh.dlb` `libcppehmt.dlb`	
C++ run-time support library with exception handling	`libcpprteh.dlb` `libcpprtehmt.dlb`	
C++ exception handling support library	`libeh.dlb` `libehmt.dlb`	
DSP run-time library	`libdsp.dlb` `libdsp020.dlb`	ADSP-21020 processor only

I/O run-time library	`libio.dlb`
	`libio020.dlb`
	`libiomt.dlb`
		ADSP-21020 processor only

I/O run-time library with no support for alternative device drivers or printf(“%a”)	`libio_lite.dlb`
	`libio020_lite.dlb`
	`libio_litemt.dlb`
	
	`libio32.dlb`
	`libio64.dlb`
		ADSP-21020 processor only
		
		Legacy library
		Legacy library

Table 3-1. C and C++ Files and Libraries for ADSP-210xx Processors

Description	Library Name	Comments

C start-up file — calls set-up routines and `main()`	`020_hdr.doj`
	`060_hdr.doj`
	
	`061_hdr.doj`
	`065L_hdr.doj`
		ADSP-21020 processor only
		ADSP-21060/062 processors only
		ADSP-21061 processor only
		ADSP-21065L processor only

C start-up file with EZ-kit — calls set-up routines and `main()`	`061_hdr_ezkit.doj` `065L_hdr_ezkit.doj`	ADSP-21061 processor only ADSP-21065L processor only

C++ start-up file — calls set-up routines and `main()`	`060_cpp_hdr.doj`
	
	`061_cpp_hdr.doj`
	`065L_cpp_hdr.doj`
	
	`060_cpp_hdr_mt.doj`
	
	`061_cpp_hdr_mt.doj`
	`065L_cpp_hdr_mt.doj`
		ADSP-21060/062 processors only
		ADSP-21061 processor only
		ADSP-21065L processor only
		
		ADSP-21060/062 processors only
		ADSP-21061 processor only
		ADSP-21065L processor only

C++ start-up file with EZ-kit — calls set-up routines and `main()`	`061_cpp_hdr_ezkit.doj`
	`065L_cpp_hdr_ezkit.doj`
	
	`061_cpp_hdr_ezkit_mt.doj`
	`065L_cpp_hdr_ezkit_mt.doj`
		ADSP-21061 processor only
		ADSP-21065L processor only
		
		ADSP-21061 processor only
		ADSP-21065L processor only

The binary files that have been built for ADSP-2116x processors are catalogued in Table 3-2.

Those components that have been built for the ADSP-21160 processors have been compiled with the `-workaround rframe` compiler switch (on page 1-56) which avoids the processor's `RFRAME` anomaly, while those components built for the ADSP-21161 processor have been compiled with the `-workaround 21161-anomaly-45` switch (on page 1-56) and thus work around the anomaly associated with conditional `DAG1` instructions. These binaries are installed in the subdirectory `211xx\lib`.

An additional set of library components is available in the subdirectory `211xx\lib\swfa` that also avoid the shadow write FIFO anomaly that affects ADSP-2116x processor architectures. These library components are selected automatically when compiling with the `-workaround swfa` switch (on page 1-56).

Table 3-2. C and C++ Files and Libraries for ADSP-2116x Processors

Description	Library Name	Comments

C run-time library	`libc160.dlb`
	`libc161.dlb`
	
	`libc160mt.dlb`
	`libc161mt.dlb`
		ADSP-21160 processor only
		ADSP-21161 processor only
		ADSP-21160 processor only
		ADSP-21161 processor only

C++ run-time library g	`libcpp.dlb` `libcppmt.dlb`	
C++ run-time support library	`libcpprt.dlb` `libcpprtmt.dlb`	
C++ run-time library with exception handling	`libcppeh.dlb` `libcppehmt.dlb`	
C++ run-time support library with exception handling	`libcpprteh.dlb` `libcpprtehmt.dlb`	
C++ exception handling support library	`libeh.dlb` `libehmt.dlb`	
DSP run-time library	`libdsp160.dlb`	
I/O run-time library	`libio.dlb` `libiomt.dlb`	

I/O run-time library with no support for alternative device drivers or printf("%a")	`libio_lite.dlb`
	`libio_litemt.dlb`
	
	`libio32.dlb`
	`libio64.dlb`
		
		Legacy library
		Legacy library

C start-up file — calls set-up routines and `main()`	`160_hdr.doj` `161_hdr.doj`	ADSP-21160 processor only ADSP-21161 processor only

Table 3-2. C and C++ Files and Libraries for ADSP-2116x Processors

Description	Library Name	Comments
C start-up file with EZ-kit — calls set-up routines and `main()`	`160_hdr_ezkit.doj`	ADSP-21160 processor only

C++ start-up file — calls set-up routines and `main()`	`160_cpp_hdr.doj`
	`161_cpp_hdr.doj`
	
	`160_cpp_hdr_mt.doj`
	`161_cpp_hdr_mt.doj`
		ADSP-21160 processor only
		ADSP-21161 processor only
		
		ADSP-21160 processor only
		ADSP-21161 processor only

C++ start-up file with EZ-kit — calls set-up routines and `main()`	`160_cpp_hdr_ezkit.doj` `160_cpp_hdr_ezkit_mt.doj`	ADSP-21160 processor only ADSP-21160 processor only

The run-time libraries and binary files for the ADSP-21160 processors in this table have been compiled with the `-workaround rframe` compiler switch , while those for the ADSP-21161 processors have been compiled with the `-workaround 21161-anomaly-45` switch. An additional set of libraries and binary files that also work around the shadow write FIFO anomaly that affect ADSP-2116x chips is installed in the subdirectory `211xx\lib\swfa`.

Table 3-3 contains a list and a brief description of the library files that have been built for the ADSP-212xx processors. These files are installed in the subdirectory `212xx\lib`.

Table 3-3. C/C++ Files and Libraries for ADSP-212xx Processors

Description	Library Name	Comments
C run-time library	`libc26x.dlb` `libc26xmt.dlb`	
C++ run-time library g	`libcpp.dlb` `libcppmt.dlb`	
C++ run-time support library	`libcpprt.dlb` `libcpprtmt.dlb`	
C++ run-time library with exception handling	`libcppeh.dlb` `libcppehmt.dlb`	

Table 3-3. C/C++ Files and Libraries for ADSP-212xx Processors (Cont'd)

Description	Library Name	Comments
C++ run-time support library with exception handling	libcpprteh.dlb libcpprtehmt.dlb	
C++ exception handling support library	libeh.dlb libehmt.dlb	
DSP run-time library	libdsp26x.dlb	
I/O run-time library	libio.dlb libiomt.dlb	
I/O run-time library with no support for alternative device drivers or printf("%a")	libio_lite.dlb libio_litemt.dlb	
C start-up file — calls set-up routines and main()	261_hdr.doj 262_hdr.doj 266_hdr.doj 267_hdr.doj	ADSP-21261 processor only ADSP-21262 processor only ADSP-21266 processor only ADSP-21267 processor only
C++ start-up file — calls set-up routines and main()	261_cpp_hdr.doj 262_cpp_hdr.doj 266_cpp_hdr.doj 267_cpp_hdr.doj 261_cpp_hdr_mt.doj 262_cpp_hdr_mt.doj 266_cpp_hdr_mt.doj 267_cpp_hdr_mt.doj	ADSP-21261 processor only ADSP-21262 processor only ADSP-21266 processor only ADSP-21267 processor only ADSP-21261 processor only ADSP-21262 processor only ADSP-21266 processor only ADSP-21267 processor only

The libraries located in 212xx\lib are built without any workarounds enabled. There are directories within the 212xx\lib directory named 2126x_rev_<revision> that contain libraries built for that specific revision, for example, 2126x_rev_0.0. A single revision library directory may support more than one specific silicon revision; as an example, 2126x_rev_0.0 supports revisions 0.0, 0.1 and 0.2 of ADSP-2126x processors.

In addition, a library directory called `2126x_any` is supplied. Libraries in this directory will contain workarounds for all relevant anomalies on all revisions of ADSP-2126x processors.

The `-si-revision` switch (on page 1-49) can be used to specify a silicon revision—VisualDSP++ will use the appropriate libraries to build the application.

Table 3-4 describes the library files that have been built for the ADSP-213xx processors, and which are installed in the subdirectory `213xx\lib`.

Table 3-4. C/C++ Files and Libraries for ADSP-213xx Processors

Description	Library Name	Comments
C run-time library	`libc36x.dlb` `libc36xmt.dlb`	
C++ run-time library g	`libcpp.dlb` `libcppmt.dlb`	
C++ run-time support library	`libcpprt.dlb` `libcpprtmt.dlb`	
C++ run-time library with exception handling	`libcppeh.dlb` `libcppehmt.dlb`	
C++ run-time support library with exception handling	`libcpprteh.dlb` `libcpprtehmt.dlb`	
C++ exception handling support library	`libeh.dlb` `libehmt.dlb`	
DSP run-time library	`libdsp36x.dlb`	
I/O run-time library	`libio.dlb` `libiomt.dlb`	
I/O run-time library with no support for alternative device drivers or `printf("%a")`	`libio_lite.dlb` `libio_litemt.dlb`	

Table 3-4. C/C++ Files and Libraries for ADSP-213xx Processors (Cont'd)

Description	Library Name	Comments

C start-up file — calls set-up routines and `main()`	`363_hdr.doj`
	`364_hdr.doj`
	`365_hdr.doj`
	`366_hdr.doj`
	`367_hdr.doj`
	`368_hdr.doj`
	`369_hdr.doj`
		ADSP-21363 processor only
		ADSP-21364 processor only
		ADSP-21365 processor only
		ADSP-21366 processor only
		ADSP-21367 processor only
		ADSP-21368 processor only-
		ADSP-21369 processor only

C++ start-up file — calls set-up routines and `main()`	`363_cpp_hdr.doj`
	`364_cpp_hdr.doj`
	`365_cpp_hdr.doj`
	`366_cpp_hdr.doj`
	`367_cpp_hdr.doj`
	`368_cpp_hdr.doj`
	`369_cpp_hdr.doj`
	
	`363_cpp_hdr_mt.doj`
	`364_cpp_hdr_mt.doj`
	`365_cpp_hdr_mt.doj`
	`366_cpp_hdr_mt.doj`
	`367_cpp_hdr_mt.doj`
	`368_cpp_hdr_mt.doj`
	`369_cpp_hdr_mt.doj`
		ADSP-21363 processor only
		ADSP-21364 processor only
		ADSP-21365 processor only
		ADSP-21366 processor only
		ADSP-21367 processor only
		ADSP-21368 processor only
		ADSP-21369 processor only
		
		ADSP-21363 processor only
		ADSP-21364 processor only
		ADSP-21365 processor only
		ADSP-21366 processor only
		ADSP-21367 processor only
		ADSP-21368 processor only
		ADSP-21369 processor only

When you call a run-time library function, the call creates a reference that the linker resolves when linking your program. One way to direct the linker to the library's location is to use the default Linker Description File (`ADSP-<your_target>.ldf`).

If you are not using the default `.LDF` file, then either add the appropriate library/libraries to the `.LDF` file used for your project, or use the compiler's `-l` switch to specify the library to be added to the link line. For example, the switches `-lc -ldsp` add `libc.dlb` and `libdsp.dlb` to the list of libraries to be searched by the linker. For more information on the `.LDF` file, see the *VisualDSP++ 4.0 Linker and Utilities Manual*.

Working with Library Header Files

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

A list of the header files that are supplied with this release of the `cc21k` compiler appears in Table 3-5. You should use a C standard text to augment the information supplied in this chapter.

Table 3-5. Standard C Run-Time Library Header Files

Header	Purpose	Standard
`assert.h`	Diagnostics	ANSI
`ctype.h`	Character Handling	ANSI
`cycle_count.h`	Basic Cycle Counting	Analog extension
`cycles.h`	Cycle Counting with Statistics	Analog extension
`device.h`	Macros and data structures for alternative device drivers	Analog extension
`device_int.h`	Enumerations and prototypes for alternative device drivers	Analog extension
`errno.h`	Error Handling	ANSI
`float.h`	Floating Point	ANSI
`iso646.h`	Boolean Operators	ANSI
`limits.h`	Limits	ANSI
`locale.h`	Localization	ANSI
`math.h`	Mathematics	ANSI
`setjmp.h`	Non-Local Jumps	ANSI
`signal.h`	Signal Handling	ANSI
`stdarg.h`	Variable Arguments	ANSI

Table 3-5. Standard C Run-Time Library Header Files (Cont'd)

Header	Purpose	Standard
stddef.h	Standard Definitions	ANSI
stdio.h	Input/Output	ANSI
stdlib.h	Standard Library	ANSI
string.h	String Handling	ANSI
time.h	Date and Time	ANSI

The following sections provide 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.

cycle_count.h

The cycle_count.h header file provides an inexpensive method for benchmarking C-written source by defining basic facilities for measuring cycle counts. The facilities provided are based upon two macros, and a data type which are described in more detail in the section "Measuring Cycle Counts" on page 3-40.

cycles.h

The cycles.h header file defines a set of five macros and an associated data type that may be used to measure the cycle counts used by a section of C-written source. The macros can record how many times a particular

piece of code has been executed and also the minimum, average, and maximum number of cycles used. The facilities that are available via this header file are described in the section "Measuring Cycle Counts" on page 3-40.

device.h

The `device.h` header file provides macros and defines data structures that an alternative device driver would require to provide file input and output services for `stdio` library functions. Normally, the `stdio` functions use a default driver to access an underlying device, but alternative device drivers may be registered that may then be used transparently by these functions. This mechanism is described in "Extending I/O Support To New Devices" on page 3-50.

device_int.h

The `device_int.h` header file contains function prototypes and provides enumerations for alterative device drivers. An alternative device driver is normally provided by an application and may be used by the `stdio` library functions to access an underlying device; an alternative device driver may coexist with, or may replace, the default driver that is supported by the VisualDSP++ simulator and EZ-KIT Lite evaluation systems. Refer to "Extending I/O Support To New Devices" on page 3-50 for more information.

errno.h

The `errno.h` header file provides access to `errno` and also defines macros for associated error codes. This facility is not, in general, supported by the rest of the library.

float.h

The `float.h` header file defines the properties of the floating-point data types that are implemented by the compiler—that is, `float`, `double`, and `long double`. These properties are defined as macros and include the following for each supported data type:

- the maximum and minimum value (for example, `FLT_MAX` and `FLT_MIN`)
- the maximum and minimum power of ten (for example, `FLT_MAX_10_EXP` and `FLT_MIN_10_EXP`)
- the precision available expressed in terms of decimal digits (for example, `FLT_DIG`)
- a constant that represents the smallest value that may added to 1.0 and still result in a change of value (for example, `FLT_EPSILON`)

Note that the set of macros that define the properties of the `double` data type will have the same values as the corresponding set of macros for the `float` type when `doubles` are defined to be 32 bits wide, and they will have the same value as the macros for the `long double` data type when `doubles` are defined to be 64 bits wide (see "-double-size[-32|-64]" on page 1-27).

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 3-6.

Table 3-6. Symbolic Names Defined in iso646.h

Symbolic Name	Equivalent
`and`	&&
`and_eq`	&=

Table 3-6. Symbolic Names Defined in iso646.h (Cont'd)

Symbolic Name	Equivalent
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

The symbolic names have the same name as the C++ keywords that are accepted by the compiler when the `-alttok` switch (see on page 1-24) 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 file 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 arranges that all references to the `double` functions are directed to the equivalent `float` function (with the suffix `f`). This allows you to 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`. Refer to Chapter 4, "DSP Library for ADSP-2106x and ADSP-21020 Processors", and Chapter 5, "DSP Library for ADSP-2116x/2126x/2136x Processors", for more information about these additional functions.

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

The macros `EDOM` and `ERANGE`, defined in `errno.h`, are used by `math.h` functions to indicate domain and range errors.

A domain error occurs when an input argument is outside the domain of the function. "C Run-Time Library Reference" (starting on page 3-64) lists the specific cases that cause `errno` to be set to `EDOM`, and the associated return values.

A range error occurs when the result of a function cannot be represented in the return type. If the result overflows, the function returns the value `HUGE_VAL` with the appropriate sign. If the result underflows, the function returns a zero without indicating a range error.

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 extensions, such as `interrupt()` and `clear_interrupt()`.

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

Interrupt Dispatchers

There are currently five types of the interrupt dispatcher, each providing different levels of functionality and performance. Each of the dispatchers is discussed in turn, starting with the slowest and most comprehensive. Note that for each dispatcher, two variants of the set-up functions are available: one which uses self-modifying code and one which does not. The non-self-modifying variants are discussed at the end of this section.

For the **circular buffer interrupt dispatcher**, use the `interruptcb()` or `signalcb()` functions to set up the interrupt. This dispatcher provides the following services:

- Saves all Data registers, Index registers, Modify registers; saves all relevant Length registers and zeroes them before calling the ISR; saves the volatile Base registers; save the contents of the loop counter stack, meaning that `DO` loops can be used safely in the ISR (interrupt service routine). On platforms with a second processing element, the S registers are also saved.
- Sends the interrupt number to the ISR as a parameter.
- On ADSP-2106x processors, requires approximately 176 cycles before calling the dispatcher and 106 cycles to return to the interrupted code; on ADSP-2116x/2126x/2136x processors, requires approximately 211 cycles before calling the dispatcher and 121 cycles to return to the interrupted code.
- Interrupt nesting is allowed.

For the **normal interrupt dispatcher**, use the `interrupt()` or `signal()` functions to set up the interrupt. This dispatcher provides the following services:

- Saves all Data registers, Index registers, Modify registers; saves the volatile Base registers; save the contents of the loop counter stack, meaning that `DO` loops can be used safely in the ISR (interrupt service routine). On platforms with a second processing element (ADSP-2116x, ADSP-2126x, and ADSP-2136x processors), the S registers are also saved.
- On ADSP-2106x processors, requires approximately 148 cycles before calling the dispatcher and 90 cycles to return to the interrupted code; on ADSP-2116x/2126x/2136x processors, requires approximately 183 cycles before calling the dispatcher and 109 cycles to return to the interrupted code.
- Sends the interrupt number type to the ISR as a parameter
- Interrupt nesting is allowed.

For the **fast interrupt dispatcher**, use the `interruptf()` or `signalf()` functions. This dispatcher provides the following services:

- Saves all scratch registers. Does not save the loop stack, therefore `DO` loop handling is restricted to 6 levels in total (specified in hardware). If the ISR uses one level of nesting, your code must not use more than five levels. The `-restrict-hardware-loops` switch (on page 1-47) controls the level of loop nesting that a function is using.
- Interrupt nesting is allowed.
- Does not send the interrupt number type to the ISR as a parameter.
- On ADSP-2106x processors, requires approximately 45 cycles before calling the dispatcher and 36 cycles to return to the interrupted code; on ADSP-2116x/2126x/2136x processors, requires approximately 40 cycles before calling the dispatcher and 26 cycles to return to the interrupted code.

For the **super-fast interrupt dispatcher**, use the `interrupts()` or `signals()` functions. This dispatcher provides the following services:

- Does not save the loop stack, therefore `DO` loop handling is restricted to six levels (specified in hardware). Interrupt nesting is disabled. This dispatcher does not send the interrupt number type to the ISR as a parameter.
- Uses the secondary register set. As a result, interrupt nesting is disabled while the dispatcher and ISR are being executed.
- On ADSP-2106x processors, requires approximately 36 cycles before calling the dispatcher and 11 cycles to return to the interrupted code; on ADSP-2116x/2126x/2136x processors, requires approximately 34 cycles before calling the dispatcher and 10 cycles to return to the interrupted code.

The **final interrupt dispatcher** is intended for use with user-written assembly functions or C functions that have been compiled using "`#pragma interrupt`" (see "Interrupt Handler Pragma" on page 1-115). Use the `interruptss()` or `signalss()` function to utilize this dispatcher. This dispatcher provides the following services:

- Relies on the compiler (or assembly routine) to save and restore all appropriate registers.
- Does not save the loop stack, therefore DO loop handling is restricted to six levels (specified in hardware).
- Interrupt nesting is allowed.
- On ADSP-2106x processors, requires approximately 29 cycles before calling the dispatcher and 24 cycles to return to the interrupted code; on ADSP-2116x/2126x/2136x processors, requires approximately 24 cycles before calling the dispatcher and 15 cycles to return to the interrupted code.

Interrupts and Circular Buffering

Only the circular buffer interrupt dispatcher and pragma interrupt dispatcher ensure that all `Length` registers are zeroed at the start of the interrupt handler. Since the compiler can generate circular buffer code automatically, you should ensure that you choose the correct dispatcher for your application (refer to "Interrupt Dispatchers" on page 3-18). Use the `-no-circbuf` switch (on page 1-38) to disable the automatic circular buffer code generation feature.

Avoiding Self-Modifying Code

The interrupt set-up functions use self-modifying code to set up the interrupts as this offers savings in execution time and code size.
Non-self-modifying variants of all these functions are supplied and are suffixed with "`nsm`". For example, to utilize the fast interrupt dispatcher,

use the `interruptfnsm()` or `signalfnsm()` functions. The choice of a non-self-modifying function has no effect on the dispatcher used and no effect on the overall interrupt handling performance.

Interrupt Nesting Restrictions on ADSP-2116x/2126x/2136x Processors

For ADSP-2116x/2126x/2136x processors, the following restrictions exist:

On these platforms, the interrupt vector code explicitly saves the `ASTATx`, `ASTATy` and `MODE1` registers on the status stack using a `push STS` instruction. For the timer, `VIRPT` and `IRQ0-2` interrupts, the save occurs in addition to the automatic save of these registers. This has the effect of reducing the maximum depth of nested interrupts to between 10 and 15 levels, depending on whether the timer, `VIRPT` and `IRQ0-2` interrupts are used.

Restriction on Use of Super-Fast Dispatcher on ADSP-2106x Processors

One of the interrupt dispatchers supplied with VisualDSP++, `interrupts()`, relies on the use of the alternate register set. Therefore, it is not possible to service another interrupt while the current interrupt is being serviced. To ensure this action, the interrupt enable bit (`IRPTEN`) is cleared by the `interrupts()` dispatcher while it is servicing an interrupt, and then is restored at the end.

Under certain, unusual circumstances on the ADSP-2106x processors, the interrupt enable bit can be set while the `interrupts()` dispatcher is being executed. A nested interrupt can result, possibly causing data corruption or other problems in user code.

The problem occurs when a lower priority interrupt (LPI) occurs and then, immediately afterwards, a higher priority interrupt (HPI) occurs. The problem only appears if the LPI is being serviced using `interrupts()`. When the LPI occurs, the processor jumps to the appropriate point in the interrupt vector table and executes the relevant code. In the default vector table, the first instruction disables the `IRPTEN` register, stopping any fur-

ther interrupts from being serviced. If, however, a HPI occurs "immediately" after the LPI (before IRPTEN is disabled), the service routine of the higher priority is executed. There is a 1-cycle delay to allow the first instruction of the lower-priority service routine (the clearing of IRPTEN) to be executed. When the HPI completes, it sets IRPTEN and returns control to the LPI. The LPI executes, the IRPTEN bit is set, and nested interrupts are able to take place. Here is a brief summary of the sequence of events which can cause this problem:

1. Lower priority interrupt (LPI) occurs
2. Higher priority interrupt (HPI) occurs
3. First instruction of LPI is executed and clears IRPTEN
4. HPI is executed:
    - clear the IRPTEN register
    - execute handler and set the IRPTEN register
5. LPI is executed:
    - The IRPTEN register was reset after HPI and is now set
    - problems may appear

Note that this is a problem on ADSP-2106x processors only.

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 `libio.dlb` in the same way as linking with the C run-time library (see "Linking Library Functions" on page 3-4). 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 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 results 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. By default, these operations are provided by the VisualDSP++ simulator and EZ-KIT Lite systems and this mechanism is outlined in section "Default Device Driver Interface" on page 3-58.

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 3-50 for a description of the feature. Applications that do not require this functionality may be built with the `-flags-link -MD__LIBIO_LITE` switch (on page 1-29). The switch links the application with a version of the I/O library that does not support the ability to register alternative device drivers and does not support the `%a` conversion specifier in `printf`. Linking with this switch results in a smaller executable.

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`
- Support for formatted reading and writing of data of `long double` type is only supported if an application is built with the `-double-size-64` switch

At program termination, the host environment closes down any physical connection between the application and an opened file. However, the I/O library does 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`, and `fprintf`, is not automatically flushed. Applications should therefore arrange to close down any streams that they open. Note that the function reference `fflush(NULL);` flushes 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 resides at a memory location that is less than `0x40000000`.

stdlib.h

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

This header file also provides prototypes for a number of additional integer math functions provided by Analog Devices, such as `avg`, `max`, and `clip`. Table 3-7 is a summary of the additional library functions defined by the `stdlib.h` header file.

Some functions exist as both integer and floating point functions. The floating-point functions typically have an `f` prefix. Make sure you use the correct type.

Table 3-7. Standard Library - Additional Functions

Description	Prototype
Average	`int avg (int a, int b);` `long lavg (long a, long b);`
Clip	`int clip (int a, int b);` `long lclip (long a, long b);`

Table 3-7. Standard Library - Additional Functions (Cont'd)

Description	Prototype
Count bits set	`int count_ones (int a);` `int lcount_ones (long a);`
Maximum	`int max (int a, int b);` `long lmax (long a, long b);`
Minimum	`int min (int a, int b);` `long lmin (long a, long b);`

Multiple heaps for dynamic memory allocation	`void *heap_calloc(int heap_index, size_t nelem, size_t size);`
	`void heap_free(int heap_index, void *ptr);`
	`void *heap_malloc(int heap_index, size_t size);`
	`void *heap_realloc(int heap_index, void *ptr, size_t size);`
	`int set_alloc_type(char * heap_name);`
	`int heap_install(void *base, size_t size, int userid, int pmdm);`
	`int heap_lookup_name(char *userid);`

A number of functions, including `abs`, `avg`, `max`, `min`, and `clip`, are implemented via intrinsics (provided the header file has been `#include`'d) that map to single-cycle machine instructions.

If the header file 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 provides functions, data types, and a macro for expressing and manipulating date and time information. The header file defines two fundamental data types, one of which is `clock_t` and is associated with the number of implementation-dependent processor "ticks" used since an arbitrary starting point; and the other which is `time_t`.

The time_t data type is used for values that represent the number of seconds that have elapsed since a known epoch; values of this form are known as a *calendar time*. In this implementation, the epoch starts on 1st January, 1970, and calendar times before this date are represented as negative values.

A calendar time may also be represented in a more versatile way as a broken-down time which is a structured variable of the following form:

```
struct tm { int tm_sec;  /* seconds after the minute [0,61] */
  int tm_min;           /* minutes after the hour [0,59]   */
  int tm_hour;          /* hours after midnight [0,23]     */
  int tm_mday;          /* day of the month [1,31]         */
  int tm_mon;           /* months since January [0,11]     */
  int tm_year;          /* years since 1900                */
  int tm_wday;          /* days since Sunday [0, 6]        */
  int tm_yday;          /* days since January 1st [0,365]  */
  int tm_isdst;         /* Daylight Saving flag            */
};
```

This implementation does not support either the Daylight Saving flag in the structure struct tm; nor does it support the concept of time zones. All calendar times are therefore assumed to relate to Greenwich Mean Time (Coordinated Universal Time or UTC).

The header file sets the CLOCKS_PER_SEC macro to the number of processor cycles per second and this macro can therefore be used to convert data of type clock_t into seconds, normally by using floating-point arithmetic to divide it into the result returned by the clock function.

In general, the processor speed is a property of a particular chip and it is therefore recommended that the value to which this macro is set is verified independently before it is used by an application.

In this version of the C/C++ compiler, the CLOCKS_PER_SEC macro is set by one of the following (in descending order of precedence):

- Via the -DCLOCKS_PER_SEC=<definition> compile-time switch

- Via the **Processor speed** box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Processor** category
- From the header file `cycles.h`

Calling Library Functions from an ISR

Not all C run-time library functions are interrupt-safe (and can therefore be called from an Interrupt Service Routine). For a run-time function to be classified as *interrupt-safe*, it must:

- not update any global data, such as `errno`, and
- not write to (or maintain) any private static data

It is recommended therefore that none of the functions defined in the header file `math.h`, nor the string conversion functions defined in `stdlib.h`, be called from an ISR as these functions are commonly defined to update the global variable `errno`. Similarly, the functions defined in the `stdio.h` header file maintain static tables for currently opened streams and should not be called from an ISR.

Several other library functions are not interrupt-safe because they make use of private static data. These functions are:

```
asctime
gmtime
localtime
rand
srand
strtok
```

While not all C run-time library functions are interrupt-safe, versions of the functions are available that are *thread-safe* and may be used in a VDK multi-threaded environment. These library functions can be found in the run-time libraries that have the suffix `_mt` in their filename.

Using Compiler Built-In C Library Functions

The C compiler intrinsic (built-in) functions are functions that the compiler immediately recognizes and replaces with inline assembly code instead of a function call. For example, the absolute value function, `abs()`, is recognized by the compiler, which subsequently replaces a call to the C run-time library version with an inline version. The `cc21k` compiler contains a number of intrinsic built-in functions for efficient access to various features of the hardware.

Built-in functions are recognized for cases where the name begins with the string `__builtin`, and the declared prototype of the function matches the prototype that the compiler expects. Built-in functions are declared in system header files. Include the appropriate header file in your program to use these functions. The normal action of the appropriate include file is to `#define` the normal name as mapping to the built-in form.

Typically, inline assembly code is faster than an average library routine, and it does not incur the calling overhead. The routines in Table 3-8 are built-in C library functions for the `cc21k` compiler:

Table 3-8. Compiler Built-in Functions

`abs`	`avg`	`clip`
`copysign`[1]	`copysignf`	`fabs`[1]
`fabsf`	`favg`[1]	`favgf`
`fclip`[1]	`fclipf`	`fmax`[1]
`fmaxf`	`fmin`[1]	`fminf`
`labs`	`lavg`	`lclip`
`lmax`	`lmin`	`max`
`min`		

1 These functions are only compiled as a built-in function if `double` is the same size as `float`.

If you want to use the C run-time library functions of the same name, compile with the `-no-builtin` compiler switch (see on page 1-37).

For a certain category of library function, the compiler relaxes the normal rule whereby pointers that are passed as arguments must address Data Memory (DM). For functions in this category, any argument that is a pointer may also address Program Memory (PM). When the compiler recognizes that certain arguments reference PM, it generates a call to an appropriate version of the function in the run-time library.

Table 3-9 contains a list of library functions that may be called with pointers to Program Memory. Note that this facility is only available provided that the `-no-builtin` compiler switch has not been specified.

Table 3-9. Dual Memory Capable Functions

atof	atoi	atol	frexp
frexpf	memchr	memcmp	memcpy
memmove	memset	modf	modff
setlocale	strcat	strchr	strcmp
strcoll	strcpy	strcspn	strlen
strncat	strncmp	strncpy	strpbrk
strrchr	strspn	strstr	strtod
strtok	strtol	strtoul	strxfrm

Abridged C++ Library Support

When in C++ mode, the `cc21k` compiler can call a large number of functions from the Abridged Library, a conforming subset of C++ library.

C++ is not supported for ADSP-21020 processors.

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 3-32
- "C++ Header Files for C Library Facilities" on page 3-35
- "Embedded Standard Template Library Header Files" on page 3-37
- "Using the Thread-Safe C/C++ Run-Time Libraries with VDK" on page 3-39

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

Embedded C++ Library Header Files

The following section provides a brief description of the header files in the embedded C++ library.

complex

The `complex` header file defines a template class `complex` and a set of associated arithmetic operators. Predefined types include `complex_float` and `complex_long_double`.

This implementation does not support the full set of complex operations as specified by the C++ standard. In particular, it does not support either the transcendental functions or the I/O operators << and >>.

The `complex` header and the C library header file `complex.h` refer to two different and incompatible implementations of the complex data type.

exception

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

fract

The `fract` header file 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 1-146. An example that demonstrates its use appears under "C++ Programming Examples" on page 1-223.

fstream

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

iomanip

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

ios

The `ios` header file defines several classes and functions for basic `iostream` manipulations. Note that most of the `iostream` header files include `ios`.

iosfwd

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

iostream

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

istream

The `istream` header file defines the `istream` class for `iostream` extractions. Note that most of the `iostream` header files include `istream`.

new

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

ostream

The `ostream` header file defines the `ostream` class for `iostream` insertions.

sstream

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

stdexcept

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

streambuf

The `streambuf` header file defines the `streambuf` classes for basic operations of the `iostream` classes. Note that most of the `iostream` header files include `streambuf`.

string

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

C++ Header Files for C Library Facilities

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

Table 3-10 lists the C++ header files that provide access to the C library facilities.

Normally, the C standard headers files may be used to define names in the C++ global namespace while the equivalent C++ header files define names in the standard namespace. However, the standard namespace is not supported in this release of the compiler, and the effect of including one of the C++ header files, listed in Table 3-10, is the same as including the equivalent C standard library header file. When porting source written to

use namespace std it may be necessary to exclude using statements and "`std::`" references. A useful way of removing `std::` references is to define `std` as a macro to nothing (for example, by using the switch `-Dstd=""`).

Table 3-10. C++ Header Files for C Library Facilities

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

Chapters 4 and 5 describe the functions in the DSP run-time libraries. Referencing these functions with a namespace prefix is not supported. All DSP library functions are in the global namespace.

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 `cc21k` compiler in C++ mode. The embedded standard template library header files are:

algorithm

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

deque

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

functional

The `functional` header file defines numerous function objects.

hash_map

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

hash_set

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

iterator

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

list

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

map

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

memory

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

numeric

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

queue

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

set

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

stack

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

utility

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

vector

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

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

fstream.h

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

iomanip.h

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

iostream.h

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

new.h

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

Using the Thread-Safe C/C++ Run-Time Libraries with VDK

When developing for VDK, the thread-safe variants of the run-time libraries are linked with user applications. These libraries may add an overhead to the VDK resources required by some applications.

The run-time libraries make use of VDK semaphores and unscheduled regions to ensure thread safety. The C and C++ libraries use a constant number of semaphores that are privately allocated by VDK, and these are in addition to any semaphores that a user may specify.

However the C++ `streambuf` class requires a semaphore for each instance of the class that is created. `Streambuf` objects are created as part of the C++ Standard I/O mechanism, and a `streambuf` may be created for each I/O stream that is created. VDK will privately allocate semaphores for the standard I/O streams `cout`, `cerr` and `cin`, and an additional semaphore is allocated for I/O accesses within the standard C library.

Users should be aware that they would need to increase the maximum semaphore count for their project if they create any additional `streambuf` objects.

Measuring Cycle Counts

The common basis for benchmarking some arbitrary C-written source is to measure the number of processor cycles that the code uses. Once this figure is known, it can be used to calculate the actual time taken by multiplying the number of processor cycles by the clock rate of the processor. The run-time library provides three alternative methods for measuring processor counts. Each of these methods is described in:

- "Basic Cycle Counting Facility" on page 3-40
- "Cycle Counting Facility with Statistics" on page 3-42
- "Using time.h to Measure Cycle Counts" on page 3-45
- "Determining the Processor Clock Rate" on page 3-47
- "Considerations When Measuring Cycle Counts" on page 3-47

Basic Cycle Counting Facility

The fundamental approach to measuring the performance of a section of code is to record the current value of the cycle count register before executing the section of code, and then reading the register again after the code has been executed. This process is represented by two macros that are defined in the `cycle_count.h` header file; the macros are:

```
START_CYCLE_COUNT(S)
STOP_CYCLE_COUNT(T,S)
```

The parameter `S` is set by the macro `START_CYCLE_COUNT` to the current value of the cycle count register; this value should then be passed to the macro `STOP_CYCLE_COUNT`, which will calculate the difference between the

parameter and current value of the cycle count register. Reading the cycle count register incurs an overhead of a small number of cycles and the macro ensures that the difference returned (in the parameter `T`) will be adjusted to allow for this additional cost. The parameters `S` and `T` should be separate variables; they should be declared as a `cycle_t` data type which the header file `cycle_count.h` defines as:

```
typedef volatile unsigned long cycle_t;
```

The header file also defines the macro:

```
PRINT_CYCLES(STRING,T)
```

which is provided mainly as an example of how to print a value of type `cycle_t`; the macro outputs the text `STRING` on `stdout` followed by the number of cycles `T`.

The instrumentation represented by the macros defined in this section is only activated if the program is compiled with the `-DDO_CYCLE_COUNTS` switch. If this switch is not specified, then the macros are replaced by empty statements and have no effect on the program.

The following example demonstrates how the basic cycle counting facility may be used to monitor the performance of a section of code:

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

extern int
main(void)
{
   cycle_t start_count;
   cycle_t final_count;

   START_CYCLE_COUNT(start_count)
   Some_Function_Or_Code_To_Measure();
   STOP_CYCLE_COUNT(final_count,start_count)

   PRINT_CYCLES("Number of cycles: ",final_count)
}
```

The run-time libraries provide alternative facilities for measuring the performance of C source (see "Cycle Counting Facility with Statistics" on page 3-42 and "Using time.h to Measure Cycle Counts" on page 3-45); the relative benefits of this facility are outlined in "Considerations When Measuring Cycle Counts" on page 3-47.

The basic cycle counting facility is based upon macros; it may therefore be customized for a particular application if required, without the need for rebuilding the run-time libraries.

Cycle Counting Facility with Statistics

The `cycles.h` header file defines a set of macros for measuring the performance of compiled C source. As well as providing the basic facility for reading the `EMUCLK` cycle count register of the SHARC architecture, the macros also have the capability of accumulating statistics that are suited to recording the performance of a section of code that is executed repeatedly.

If the switch `-DDO_CYCLE_COUNTS` is specified at compile-time, then the `cycles.h` header file defines the following macros:

- `CYCLES_INIT(S)`
  a macro that initializes the system timing mechanism and clears the parameter `S`; an application must contain one reference to this macro.
- `CYCLES_START(S)`
  a macro that extracts the current value of the cycle count register and saves it in the parameter `S`.
- `CYCLES_STOP(S)`
  a macro that extracts the current value of the cycle count register and accumulates statistics in the parameter `S`, based on the previous reference to the `CYCLES_START` macro.

- `CYCLES_PRINT(S)`
  a macro which prints a summary of the accumulated statistics recorded in the parameter `S`.

- `CYCLES_RESET(S)`
  a macro which re-zeros the accumulated statistics that are recorded in the parameter `S`.

The parameter `S` that is passed to the macros must be declared to be of the type `cycle_stats_t`; this is a structured data type that is defined in the `cycles.h` header file. The data type has the capability of recording the number of times that an instrumented part of the source has been executed, as well as the minimum, maximum, and average number of cycles that have been used. If an instrumented piece of code has been executed for example, 4 times, the `CYCLES_PRINT` macro would generate output on the standard stream `stdout` in the form:

```
AVG   : 95
MIN   : 92
MAX   : 100
CALLS : 4
```

If an instrumented piece of code had only been executed once, then the `CYCLES_PRINT` macro would print a message of the form:

```
CYCLES : 95
```

If the switch `-DDO_CYCLE_COUNTS` is not specified, then the macros described above are defined as null macros and no cycle count information is gathered. To switch between development and release mode therefore only requires a re-compilation and will not require any changes to the source of an application.

The macros defined in the `cycles.h` header file may be customized for a particular application without the requirement for rebuilding the run-time libraries.

An example that demonstrates how this facility may be used is:

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

extern void foo(void);
extern void bar(void);

extern int
main(void)
{
   cycle_stats_t stats;
   int i;

   CYCLES_INIT(stats)

   for (i = 0; i < LIMIT; i++) {
      CYCLES_START(stats)
      foo();
      CYCLES_STOP(stats)
   }
   printf("Cycles used by foo\n");
   CYCLES_PRINT(stats)
   CYCLES_RESET(stats)

   for (i = 0; i < LIMIT; i++) {
      CYCLES_START(stats)
      bar();
      CYCLES_STOP(stats)
   }
   printf("Cycles used by bar\n");
   CYCLES_PRINT(stats)
}
```

This example might output:

```
   Cycles used by foo
      AVG   : 25454
      MIN   : 23003
      MAX   : 26295
      CALLS : 16
```

```
Cycles used by bar
   AVG   : 8727
   MIN   : 7653
   MAX   : 8912
   CALLS : 16
```

Alterative methods of measuring the performance of compiled C source are described in the sections "Basic Cycle Counting Facility" on page 3-40 and "Using time.h to Measure Cycle Counts" on page 3-45. Also refer to "Considerations When Measuring Cycle Counts" on page 3-47 which provides some useful tips with regards to performance measurements.

Using time.h to Measure Cycle Counts

The `time.h` header file defines the data type `clock_t`, the `clock` function, and the macro `CLOCKS_PER_SEC`, which together may be used to calculate the number of seconds spent in a program.

In the ANSI C standard, the `clock` function is defined to return the number of implementation dependent clock "ticks" that have elapsed since the program began, and in this version of the C/C++ compiler the function returns the number of processor cycles that an application has used.

The conventional way of using the facilities of the `time.h` header file to measure the time spent in a program is to call the `clock` function at the start of a program, and then subtract this value from the value returned by a subsequent call to the function. This difference is usually cast to a floating-point type, and is then divided by the macro `CLOCKS_PER_SEC` to determine the time in seconds that has occurred between the two calls.

If this method of timing is used by an application then it is important to note that:

- the value assigned to the macro `CLOCKS_PER_SEC` should be independently verified to ensure that it is correct for the particular processor being used (see "Determining the Processor Clock Rate" on page 3-47),

- the result returned by the `clock` function does not include the overhead of calling the library function.

A typical example that demonstrates the use of the `time.h` header file to measure the amount of time that an application takes is shown below.

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

extern int
main(void)
{
    volatile clock_t clock_start;
    volatile clock_t clock_stop;

    double secs;

    clock_start = clock();
    Some_Function_Or_Code_To_Measure();
    clock_stop = clock();

    secs = ((double) (stop_time - start_time))
           / CLOCKS_PER_SEC;
    printf("Time taken is %e seconds\n",secs);
}
```

The header files `cycles.h` and `cycle_count.h` define other methods for benchmarking an application—these header files are described in the sections "Basic Cycle Counting Facility" on page 3-40 and "Cycle Counting Facility with Statistics" on page 3-42, respectively. Also refer to "Considerations When Measuring Cycle Counts" on page 3-47 which provides some guidelines that may be useful.

Determining the Processor Clock Rate

Applications may be benchmarked with respect to how many processor cycles that they use. However, more typically applications are benchmarked with respect to how much time (for example, in seconds) that they take.

To measure the amount of time that an application takes to run on a SHARC processor usually involves first determining the number of cycles that the processor takes, and then dividing this value by the processor's clock rate. The `time.h` header file defines the macro `CLOCKS_PER_SEC` as the number of processor "ticks" per second. On an ADSP-21xxx (SHARC) architecture, it is set by the run-time library to one of the following values in descending order of precedence:

- via the compile-time switch `-DCLOCKS_PER_SEC=<definition>`.
- via the **Processor speed** box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Processor** category
- from the `cycles.h` header file

If the value of the macro `CLOCKS_PER_SEC` is taken from the `cycles.h` header file, then be aware that the clock rate of the processor will usually be taken to be the maximum speed of the processor, which is not necessarily the speed of the processor at `RESET`.

Considerations When Measuring Cycle Counts

The run-time library provides three different methods for benchmarking C-compiled code. Each of these alternatives are described in the following sections:

- "Basic Cycle Counting Facility" on page 3-40
  The basic cycle counting facility represents an inexpensive and relatively inobtrusive method for benchmarking C-written source using cycle counts. The facility is based on macros that factor-in

the overhead incurred by the instrumentation. The macros may be customized and they can be switched either or off, and so no source changes are required when moving between development and release mode. The same set of macros is available on other platforms provided by Analog Devices.

- "Cycle Counting Facility with Statistics" on page 3-42
  This is a cycle-counting facility that has more features than the basic cycle counting facility described above. It is therefore more expensive in terms of program memory, data memory, and cycles consumed. However, it does have the ability to record the number of times that the instrumented code has been executed and to calculate the maximum, minimum, and average cost of each iteration. The macros provided take into account the overhead involved in reading the cycle count register. By default, the macros are switched off, but they are switched on by specifying the `-DDO_CYCLE_COUNTS` compile-time switch. The macros may also be customized for a specific application. This cycle counting facility is also available on other Analog Devices architectures.

- "Using time.h to Measure Cycle Counts" on page 3-45
  The facilities of the `time.h` header file represent a simple method for measuring the performance of an application that is portable across a large number of different architectures and systems. These facilities are based around the `clock` function.

  The `clock` function however does not take into account the cost involved in invoking the function. In addition, references to the function may affect the code that the optimizer generates in the vicinity of the function call. This method of benchmarking may not accurately reflect the true cost of the code being measured.

  This method is more suited to benchmarking applications rather than smaller sections of code that run for a much shorter time span.

When benchmarking code, some thought is required when adding instrumentation to C source that will be optimized. If the sequence of statements to be measured is not selected carefully, the optimizer may move instructions into (and out of) the code region and/or it may re-site the instrumentation itself, thus leading to distorted measurements. It is therefore generally considered more reliable to measure the cycle count of calling (and returning from) a function rather than a sequence of statements within a function.

It is recommended that variables that are used directly in benchmarking are simple scalars that are allocated in internal memory (be they assigned the result of a reference to the `clock` function, or be they used as arguments to the cycle counting macros). In the case of variables that are assigned the result of the `clock` function, it is also recommended that they be defined with the `volatile` keyword.

The different methods presented here to obtain the performance metrics of an application are based on the `EMUCLK` register. This is a 32-bit register that is incremented at every processor cycle; once the counter reaches the value `0xffffffff` it will wrap back to zero and will also increment the `EMUCLK2` register. However, to save memory and execution time, the `EMUCLK2` register is not used by either the `clock` function or the cycle counting macros. The performance metrics therefore will wrap back to zero after approximately every 71 seconds on a 60 MHz processor.

File I/O Support

The VisualDSP++ environment provides access to files on a host system by using stdio functions. File I/O support is provided through a set of low-level primitives that implement the open, close, read, write, and seek operations. The functions defined in the `stdio.h` header file 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 ADSP-21xxx installation of VisualDSP++ in the subdirectory `...\lib\src\libio_src`.

This section describes:

- "Extending I/O Support To New Devices" on page 3-50
- "Default Device Driver Interface" on page 3-58

Refer to "stdio.h" on page 3-24 for information about the conventional C input and output facilities that are provided by the compiler.

Extending I/O Support To New Devices

The I/O primitives are implemented using an extensible device driver mechanism. The default start-up code includes a device driver that can perform I/O through the VisualDSP++ simulator and EZ-KIT Lite evaluation systems. Other device drivers may be registered and then used through the normal `stdio` functions.

DevEntry Structure

A device driver is a set of primitive functions grouped together into a `DevEntry` structure. This structure is defined in `device.h`.

```
struct DevEntry {
   int   DeviceID;
   void  *data;

   int   (*init)(struct DevEntry *entry);
   int   (*open)(const char *name, int mode);
   int   (*close)(int fd);
   int   (*write)(int fd, unsigned char *buf, int size);
   int   (*read)(int fd, unsigned char *buf, int size);
   long  (*seek)(long fd, int offset, int whence);
   int stdinfd;
   int stdoutfd;
   int stderrfd;
}

typedef struct DevEntry DevEntry;
typedef struct DevEntry *DevEntry_t;
```

The fields within the DevEntry structure have the following meanings.

DeviceID:
The DeviceID field is a unique identifier for the device, known to the user. Device IDs are used globally across an application.

data:
The data field is a pointer for any private data the device may need; it is not used by the run-time libraries.

init:
The init field is a pointer to an initialization function. The run-time library calls this function when the device is first registered, passing in the address of this structure (and thus giving the init function access to DeviceID and the field data). If the init function encounters an error, it must return -1. Otherwise, it must return a positive value to indicate success.

open:
The open field is a pointer to a function performs the "*open file*" operation upon the device; the run-time library will call this function in response to requests such as fopen(), when the device is the currently-selected default device. The name parameter is the path name to the file to be opened, and the mode parameter is a bitmask that indicates how the file is to be opened:

```
0x0001   Open file for reading
0x0002   Open file for writing
0x0004   Open file for appending
0x0008   Truncate the file to zero length, if it already exists
0x00010  Create the file, if it does not already exist
```

By default, files are opened as text streams (in which the character sequence \r\n is converted to \n when reading, and the character \n is written to the file as \r\n). A file is opened as a binary stream if the following bit value is set in the mode parameter:

```
0x0020   Open the file as a binary stream (raw mode).
```

The `open` function must return a positive "*`file descriptor`*" if it succeeds in opening the file; this file descriptor is used to identify the file to the device in subsequent operations. The file descriptor must be unique for all files currently open for the device, but need not be distinct from file descriptors returned by other devices—the run-time library identifies the file by the combination of device and file descriptor.

If the `open` function fails, it must return -1 to indicate failure.

close:
The `close` field is a pointer to a function that performs the "*`close file`*" operation on the device. The run-time library calls the `close` function in response to requests such as `fclose()` on a stream that was opened on the device. The `fd` parameter is a file descriptor previously returned by a call to the `open` function. The `close` function must return a positive value for success, and a negative value for failure.

write:
The `write` field is a pointer to a function that performs the "*`write to file`*" operation on the device. The run-time library calls the `write` function in response to requests, such as `fwrite()`, `fprintf()` and so on, that act on streams that were opened on the device. The `write` function takes three parameters:

- `fd` – this is a file descriptor that identifies the file to be written to; it will be a value that was returned from a previous call to the open function.
- `buf` – a pointer to the data to be written to the file
- `size` – the number of bytes to be written to the file

The `write` function must return one of the following values:

- A positive value from 1 to `size` inclusive, indicating how many bytes from `buf` were successfully written to the file
- Zero, indicating that the file has been closed, for some reason (for example, network connection dropped)
- A negative value, indicating an error

read:
The `read` field is a pointer to a function that performs the "*`read from file`*" operation on the device. The run-time library calls the `read` function in response to requests, such as `fread()`, `fscanf()` and so on, that act on streams that were opened on the device. The `read` function's parameters are:

- `fd` – this is the file descriptor for the file to be read
- `buf` – this is a pointer to the buffer where the retrieved data must be stored
- `size` – this is the number of (8-bit) bytes to read from the file. This must not exceed the space available in the buffer pointed to by `buf`

The `read` function must return one of the following values:

- A positive value from 1 to `size` inclusive, indicating how many bytes were read from the file into `buf`
- Zero, indicating end-of-file
- A negative value, indicating an error

The run-time library expects the `read` function to return `0xa (10)` as the newline character.

seek:
The `seek` field is a pointer to a function that performs dynamic access on the file. The run-time library calls the `seek` function in response to requests such as `rewind()`, `fseek()`, and so on, that act on streams that were opened on the device.

The `seek` function takes the following parameters:

- `fd` – this is the file descriptor for the file which will have its read/write position altered
- `offset` – this is a value that is used to determine the new read/write pointer position within the file; it is in (8-bit) bytes
- `whence` – this is a value that indicates how the `offset` parameter is interpreted:
    - 0: `offset` is an absolute value, giving the new read/write position in the file
    - 1: `offset` is a value relative to the current position within the file
    - 2: `offset` is a value relative to the end of the file

The `seek` function returns a positive value that is the new (absolute) position of the read/write pointer within the file, unless an error is encountered, in which case the `seek` function must return a negative value.

If a device does not support the functionality required by one of these functions (such as read-only devices, or stream devices that do not support seeking), the `DevEntry` structure must still have a pointer to a valid function; the function must arrange to return an error for attempted operations.

stdinfd:
The `stdinfd` field is set to the device file descriptor for `stdin` if the device is expecting to claim the `stdin` stream, or to the enumeration value `dev_not_claimed` otherwise.

stdoutfd:
The `stdoutfd` field is set to the device file descriptor for `stdout` if the device is expecting to claim the `stdout` stream, or to the enumeration value `dev_not_claimed` otherwise.

stderrfd:
The `stderrfd` field is set to the device file descriptor for `stderr` if the device is expecting to claim the `stderr` stream, or to the enumeration value `dev_not_claimed` otherwise.

Registering New Devices

A new device can be registered with the following function:

```
int add_devtab_entry(DevEntry_t entry);
```

If the device is successfully registered, the `init()` routine of the device is called, with `entry` as its parameter. The `add_devtab_entry()` function returns the `DeviceID` of the device registered.

If the device is not successfully registered, a negative value is returned. Reasons for failure include (but are not limited to):

- The `DeviceID` is the same as another device, already registered
- There are no more slots left in the device registry table
- The `DeviceID` is less than zero
- Some of the function pointers are NULL
- The device's `init()` routine returned a failure result
- The device has attempted to claim a standard stream that is already claimed by another device

Pre-Registering Devices

The library source file `devtab.c` (which can be found under a VisualDSP++ installation in the subdirectory `. . . \lib\src\libio_src`) declares the array:

```
DevEntry_t DevDrvTable[];
```

This array contains pointers to `DevEntry` structures for each device that is pre-registered, that is, devices that are available as soon as `main()` is entered, and that do not need to be registered at run-time by calling `add_devtab_entry()`. By default, the "*PrimIO*" device is registered. The `PrimIO` device provides support for target/host communication when using the simulators and the Analog Devices emulators and debug agents. This device is pre-registered, so that `printf()` and similar functions operate as expected without additional setup.

Additional devices can be pre-registered by the following process:

1. Take a copy of the `devtab.c` source file and add it to your project.
2. Declare your new device's `DevEntry` structure within the `devtab.c` file, for example,

   ```
   extern DevEntry myDevice;
   ```

3. Include the address of the `DevEntry` structure within the `DevDrvTable[]` array. Ensure that the table is null-terminated. For example,

   ```
   DevEntry_t DevDrvTable[MAXDEV] = {
   #ifdef PRIMIO
      &primio_deventry,
   #endif
      &myDevice,   /* new pre-registered device */
      0,
   };
   ```

All pre-registered devices are initialized by the run-time library when it calls the `init` function of each of the pre-registered devices in turn.

The normal behavior of the `PrimIO` device when it is registered is to claim the first three files as `stdin`, `stdout` and `stderr`. These standard streams may be re-opened on other devices at run-time by using `freopen()` to close the `PrimIO`-based streams and reopen the streams on the current default device.

To allow an alternative device (either pre-registered or registered by `add_devtab_entry()`) to claim one or all of the standard streams:

1. Take a copy of the `primiolib.c` source file, and add it to your project.
2. Edit the appropriate `stdinfd`, `stdoutfd`, and `stderrfd` file descriptors in the `primio_deventry` structure to have the value `dev_not_claimed`.
3. Ensure the alternative device's `DevEntry` structure has set the standard stream file descriptors appropriately.

Both the device initialization routines, called from the startup code and `add_devtab_entry()`, return with an error if a device attempts to claim a standard stream that is already claimed.

Default Device

Once a device is registered, it can be made the default device using the following function:

```
void set_default_io_device(int);
```

The function should be passed the `DeviceID` of the device. 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's file identifier (`fd`) returned by the `open()` function is private to the device; other devices may simultaneously have other open files that use the same identifier. An open file is uniquely identified by the combination of `DeviceID` and `fd`.

The `fopen()` function records the `DeviceID` and `fd` in the global open file table, and allocates its own internal `fid` to this combination. All future operations on the file use this `fid` to retrieve the `DeviceID` and thus direct the request to the appropriate device's primitive functions, passing the `fd` along with other parameters. Once a file has been opened by `fopen()`, the current value of `get_default_io_device()` is irrelevant to that file.

Remove and Rename Functions

The `PrimIO` device provides support for the `remove()` and `rename()` functions. These functions are not currently part of the extensible File I/O interface, since they deal purely with path names, and not with file descriptors. All calls to `remove()` and `rename()` in the run-time library are passed directly to the `PrimIO` device.

Default Device Driver Interface

The stdio functions provide access to the files on a host system through a device driver that supports a set of low-level I/O primitives. These low-level primitives are described under"Extending I/O Support To New Devices" on page 3-50. 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 `. . .\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 resides in a control block at the label _PrimIOCB. If this scheme is used, the host should arrange to intercept control when it enters the __primIO routine, and, after servicing the request, return control to the calling routine.

The second scheme involves communicating with the DSP 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 causes __primIO to assume that a host environment is present and able to communicate with the process.

If __primIO sees that __lone_SHARC is cleared, then upon entry (for example, when an I/O request is made) it sets a nonzero value into the word labeled __Godot. The __primIO routine then busy-waits until this word is reset to zero by the host. The non-zero value of __Godot raised by __primIO is the address of the I/O control block.

Data Packing For Primitive I/O

The implementation of the __primIO interface is based on a word-addressable machine, with each word comprising a fixed number of 8-bit bytes. All READ and WRITE requests specify a move of some number of 8-bit bytes, that is, the relevant fields count 8-bit bytes, not words. 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 SHARC architecture. Data packing can be changed to accomodate other 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.

Before "activating" the debugger or other host environment, an OPEN or REMOVE request may set the `fileID` field to the length of the filename to open or delete; a RENAME request may also set the field to the length of the old filename. If the `fileID` field does contain a string length, then this will be indicated in the `flags` field (see below), and the debugger or other host environment will be able to use the information to perform a batch memory read to extract the filename. If the information is not provided, then the file name has to be extracted one character at a time.

The `flags` field is a bit field containing other information for special requests. Meaningful bit values for an `OPEN` operation are:

```
M_OPENR = 0x0001      /* open for reading                    */
M_OPENW = 0x0002      /* open for writing                    */
M_OPENA = 0x0004      /* open for append                     */
M_TRUNCATE = 0x0008   /* truncate to zero length if file exists */
M_CREATE = 0x0010     /* create the file if necessary        */
M_BINARY = 0x0020     /* binary file (vs. text file)         */
M_STRLEN_PROVIDED = 0x8000  /* length of file name(s) available */
```

For a `READ` operation, the low-order four bits of the `flag` value contain the number of bytes packed into each word of the read buffer, and the rest of the value is reserved for future use.

For a `WRITE` operation, the low-order four bits of the `flag` value contain the number of bytes packed into each word of the write buffer, and the rest of the value form a bit field, for which only the following bit is currently defined:

```
M_ALIGN_BUFFER = 0x10
```

If this bit is set for a WRITE request, the WRITE operation is expected to be aligned on a processor word boundary by writing padding NULs to the file before the buffer contents are transferred.

For an OPEN, REMOVE, and RENAME operation, the debugger (or other host mechanism) has to extract the filename(s) one character at a time from the memory of the target. However, if the bit corresponding to the value M_STRLEN_PROVIDED is set, then the I/O control block contains the length of the filename(s) and the debugger is able to use this information to perform a batch read of the target memory (see the description of the fields fileID and nCompleted).

For a SEEK request, the flags field indicates the seek mode (whence) as follows:

```
enum
{
   M_SEEK_SET = 0x0001,   /* seek origin is the start of
                             the file */
   M_SEEK_CUR = 0x0002,   /* seek origin is the current
                             position within the file */
   M_SEEK_END = 0x0004,   /* seek origin is the end of
                             the file */
};
```

The flags field is unused for a CLOSE request.

The buf field contains a pointer to the file name for an OPEN or REMOVE request, or a pointer to the data buffer for a READ or WRITE request. For a RENAME operation, this field contains a pointer to the old file name.

The nDesired field is set to the number of bytes that should be transferred for a READ or WRITE request. This field is also used by a RENAME request, and is set to a pointer to the new file name.

For a SEEK request, the nDesired field contains the offset at which the file should be positioned, relative to the origin specified by the flags field. (On architectures that only support 16-bit ints, the 32-bit offset at which the file should be positioned is stored in the combined fields [buf, nDesired]).

The nCompleted field is set by __primIO to the number of bytes actually transferred by a READ or WRITE operation. For a SEEK operation, __primIO sets this field to the new value of the file pointer. (On architectures that only support 16-bit ints, __primIO sets the new value of the file pointer in the combined fields [nCompleted, more]).

The RENAME operation may also make use of the nCompleted field. If the operation can determine the lengths of the old and new filenames, then it should store these sizes in the fields fileID and nCompleted, respectively, and also set the bit field flags to M_STRLEN_PROVIDED. The debugger (or other host mechanism) can then use this information to perform a batch read of the target memory to extract the filenames. If this information is not provided, then each character of the file names will have to be read individually.

The more field is reserved for future use and currently is always set to NULL before calling _primIO.

C 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.

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

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

Name and Purpose of the function

Synopsis – Required header file and functional prototype

Description – Function specification

Error Conditions – Method that the functions use to indicate an error

Example –Typical function usage

See Also – Related functions

abort

abnormal program end

Synopsis

```
#include <stdlib.h>
void abort (void);
```

Description

The `abort` function causes an abnormal program termination by raising the `SIGABRT` exception. If the `SIGABRT` handler returns, `abort()` calls `exit()` to terminate the program with a failure condition.

Error Conditions

The `abort` function does not return.

Example

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

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

See Also

atexit, exit

abs

absolute value

Synopsis

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

Description

The `abs` function returns the absolute value of its integer argument.

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

Error Conditions

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

Example

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

i  = abs (-5);      /* i == 5 */
```

See Also

fabs, labs

acos

arc cosine

Synopsis

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

Description

The acos functions return the arc cosine of `x`. The input must be in the range [-1, 1]. The output, in radians, is in the range [0, π].

Error Conditions

The acos functions indicate a domain error (set `errno` to `EDOM`) and return a zero if the input is not in the range [–1, 1].

Example

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

x = acos (0.0);       /* x = π/2 */
y = acosf (0.0);      /* y = π/2 */
```

See Also

cos

asctime

convert broken-down time into a string

Synopsis

```
#include <time.h>
char *asctime(const struct tm *t);
```

Description

The `asctime` function converts a broken-down time, as generated by the functions `gmtime` and `localtime`, into an ASCII string that will contain the date and time in the form

```
DDD MMM dd hh:mm:ss YYYY\n
```

where

- `DDD` represents the day of the week (that is. Mon, Tue, Wed, etc.)
- `MMM` is the month and will be of the form Jan, Feb, Mar, etc
- `dd` is the day of the month, from 1 to 31
- `hh` is the number of hours after midnight, from 0 to 23
- `mm` is the minute of the day, from 0 to 59
- `ss` is the second of the day, from 0 to 61 (to allow for leap seconds)
- `YYYY` represents the year

The function returns a pointer to the ASCII string, which may be over-written by a subsequent call to this function. Also note that the function `ctime` returns a string that is identical to

```
asctime(localtime(&t))
```

Error Conditions

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

Example

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

struct tm tm_date;

printf("The date is %s",asctime(&tm_date));
```

See Also

ctime, gmtime, localtime

asin

arc sine

Synopsis

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

Description

The asin functions return the arc sine of the first argument. The input must be in the range [1, 1]. The output, in radians, is in the range -π/2 to π/2.

Error Conditions

The asin functions indicate a domain error (set `errno` to `EDOM`) and return a zero if the input is not in the range [-1, 1].

Example

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

y = asin (1.0);       /* y = π/2 */
x = asinf (1.0);      /* x = π/2 */
```

See Also

sin

atan

arc tangent

Synopsis

```
#include <math.h>
double atan (double x);
float atanf (float x);
long double atand (long double x);
```

Description

The atan functions return the arc tangent of the first argument. The output, in radians, is in the range -π/2 to π/2.

Error Conditions

The atan functions do not return error conditions.

Example

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

y = atan (0.0);          /* y = 0.0 */
x = atanf (0.0);         /* x = 0.0 */
```

See Also

atan2, tan

atan2

arc tangent of quotient

Synopsis

```
#include <math.h>
double atan2 (double x, double y);
float atan2f (float x, float y);
long double atan2d (long double x, long double y);
```

Description

The atan2 functions compute the arc tangent of the input value `y` divided by input value `x`. The output, in radians, is in the range -π to π.

Error Conditions

The atan2 functions return a zero and set `errno` to `EDOM` if `x=0` and `y <> 0`.

Example

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

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

See Also

atan, tan

atexit

register a function to call at program termination

Synopsis

```
#include <stdlib.h>
int atexit (void (*func)(void));
```

Description

The `atexit` function registers a function to be called at program termination. Functions are called once for each time they are registered, in the reverse order of registration. Up to 32 functions can be registered using the `atexit` function.

Error Conditions

The `atexit` function returns a nonzero value if the function cannot be registered.

Example

```
#include <stdlib.h>
extern void goodbye(void);

if (atexit(goodbye))
   exit(1);
```

See Also

abort, exit

atof

convert string to a double

Synopsis

```
#include <stdlib.h>
double atof(const char *nptr);
```

Description

The `atof` function converts a character string into a floating-point value of type `double`, and returns its value. The character string is pointed to by the argument `nptr` and may contain any number of leading whitespace characters (as determined by the function `isspace`) followed by a floating-point number. The floating-point number may either be a decimal floating-point number or a hexadecimal floating-point number.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (–); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix `0x` or `0X`. This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P` , an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs][.hexdigs]`.

The first character that does not fit either form of number stops the scan.

Error Conditions

The `atof` function returns a zero if no conversion 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.

Example

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

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

See Also

atoi, atol, strtod

atoi

convert string to integer

Synopsis

```
#include <stdlib.h>
int atoi (const char *nptr);
```

Description

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

Error Conditions

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

Example

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

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

See Also

atof, atol, strtod, strtol, strtoul

atol

convert string to long integer

Synopsis

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

Description

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

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

Error Conditions

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

Example

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

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

See Also

atof, atoi, strtod, strtol, strtoul

atold

convert string to a long double

Synopsis

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

Description

The `atold` function converts a character string into a floating-point value of type `long double`, and returns its value. The character string is pointed to by the argument `nptr` and may contain any number of leading whitespace characters (as determined by the function `isspace`) followed by a floating-point number. The floating-point number may either be a decimal floating-point number or a hexadecimal floating-point number.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (–); and `digits` are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (-) followed by the hexadecimal prefix `0x` or `0X` . This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P` , an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan.

Error Conditions

The `atold` 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.

Example

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

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

See Also

atoi, atol, strtold

avg

mean of two values

Synopsis

```
#include <stdlib.h>
int avg (int x, int y);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `avg` function adds two arguments and divides the result by two. The `avg` function is a built-in function which is implemented with an `Rn=(Rx+Ry)/2` instruction.

Error Conditions

The `avg` function does not return an error code.

Example

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

i = avg (10, 8);      /* returns 9 */
```

See Also

lavg

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. 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.

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

Error Conditions

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

Example

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

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

See Also

qsort

calloc

allocate and initialize memory

Synopsis

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

Description

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

The object is allocated from the current heap, which is the default heap unless `set_alloc_type` has been called to change the current heap to an alternate heap.

Error Conditions

The `calloc` function returns a null pointer if unable to allocate the requested memory.

Example

```
#include <stdlib.h>
int *ptr;

ptr = (int *) calloc (10, sizeof (int));
    /* ptr points to a zeroed array of length 10 */
```

See Also

free, heap_calloc, heap_free, heap_lookup_name, heap_malloc, heap_realloc, malloc, realloc, set_alloc_type

ceil

ceiling

Synopsis

```
#include <math.h>
double ceil (double x);
float ceilf (float x);
long double ceild (long double x);
```

Description

The ceil functions return the smallest integral value that is not less than the argument `x`.

Error Conditions

The ceil functions do not return an error condition.

Example

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

y = ceil (1.05);       /* y = 2.0  */
x = ceilf (-1.05);     /* y = -1.0 */
```

See Also

floor

circindex

perform circular buffer operation on loop index

Synopsis

```
#include <21020.h>
#include <21060.h>
#include <21065l.h>
#include <21160.h>
#include <21262.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>

int circindex(int index, int incr, int num_items);
```

Description

The `circindex` function is used within a loop in order to implement a circular buffer operation in C/C++. When optimization is enabled, the operation is implemented using the appropriate hardware features (B registers and L registers) of the SHARC architecture. The `circindex` function is used to increment or decrement an index in a loop and this index should be used to access memory locations.

The argument `index` represents the index variable, `incr` represents the value by which the index should be incremented on each iteration, and `num_items` represents the size of the circular buffer.

It is not currently possible to perform circular buffer operations in SIMD mode.

Error Conditions

The `circindex` function does not return an error code.

Example

```
#include <21060.h>
#include <stdio.h>

int x[10] = {1,2,3,4,5,6,7,8,9,10};
int y[10] = {2,3,4,5,6,7,8,9,10,11};

int dot (const int *a, const int *b)
{
   int i, ci = 0;
   long s = 0;

   /* This will calculate the product for the first 5 elements
    * in each array only. As the loop count is 10, each sum will
    * be calculated twice.
    * Note that each array is indexed using 'ci'. */

   for (i = 0; i < 10; i++) {
      s += a[ci] * b[ci];
      ci = circindex(ci, 1, 5);      // Increment the index
   }

   return s;
}

void
main()
{
   int result;
   result = dot(x,y);
   printf("Result is %d\n", result);      // Result is 140
}
```

See Also

circptr

circptr

perform circular buffer operation on a pointer

Synopsis

```
#include <21020.h>
#include <21060.h>
#include <21065l.h>
#include <21160.h>
#include <21262.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>

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

Description

The `circptr` function is used within a loop in order to implement a circular buffer operation in C/C++. When optimization is enabled, the operation is implemented using the appropriate hardware features (B registers and L registers) of the SHARC processor architecture. The `circptr` function is used to increment or decrement a pointer variable in a loop.

When used with a `PM` qualified circular buffer, the result of the circular buffer function should be cast to `(void pm *)`.

The argument `ptr` represents the pointer that is being used for the circular buffer, `incr` represents the value by which the circular buffer should be incremented, `base` represents the array on which the circular buffer operates, and `buflen` represents the size of the circular buffer.

It is not currently possible to perform circular buffer operations in SIMD mode.

Error Conditions

The `circptr` function does not return an error code.

Example

```
#include <21060.h>
#include <stdio.h>

int    x[10] = {1,2,3,4,5,6,7,8,9,10};
int pm y[10] = {2,3,4,5,6,7,8,9,10,11};

int dot (const int *a, const int pm *b)
{
   int i;
   long s = 0;
   const int *cba;
   const int pm *cbb;

   /* This will calculate the product for the first 5 elements
      in each array only. As the loop count is 10,each sum will
      be calculated twice. */

   cba = a;
   cbb = b;
   for (i = 0; i < 10; i++) {
      s += *cba * *cbb;
      cba = circptr(cba, 1, a, 5);              // Increment cba
      cbb = (void pm *)circptr(cbb, 1, b, 5);   // Increment cbb
   }

   return s;
}

void
main()
{
   int result;
   result = dot(x,y);
   printf("Result is %d\n", result);             // Result is 140
}
```

See Also

circindex

clear_interrupt

clear a pending signal

Synopsis

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

Description

This function is an Analog Devices extension to the ANSI standard.

The `clear_interrupt` function clears the signal `sig` in the `IRPTL` register. Table 3-11, Table 3-12 on page 3-93, Table 3-13 on page 3-94, Table 3-14 on page 3-96, and Table 3-15 on page 3-98 show `sig` arguments of the processor signals for appropriate ADSP-21xxx processors.

The `clear_interrupt` function does not work for interrupts that set any status bits in the `STKY` register, such as floating-point overflow.

Table 3-11. ADSP-21020 Processor Signals

SIG Value	Description
SIG_SOVF	Status stack or Loop stack overflow or PC stack full
SIG_TMZ0	Timer = 0 (high priority option)
SIG_IRQ3	Interrupt 3
SIG_IRQ2	Interrupt 2
SIG_IRQ1	Interrupt 1
SIG_IRQ0	Interrupt 0
SIG_CB7	Circular buffer 7 overflow
SIG_CB15	Circular buffer 15 overflow
SIG_TMZ	Timer = 0 (low priority option)
SIG_FIX	Fixed-point overflow

Table 3-11. ADSP-21020 Processor Signals (Cont'd)

SIG_FLTO	Floating point overflow exception
SIG_FLTU	Floating point underflow exception
SIG_FLTI	Floating point invalid exception
SIG_USR0	User software interrupt 0
SIG_USR1	User software interrupt 1
SIG_USR2	User software interrupt 2
SIG_USR3	User software interrupt 3
SIG_USR4	User software interrupt 4
SIG_USR5	User software interrupt 5

Table 3-12. ADSP-2106x Processor Signals

SIG Value	**Definition**
SIG_SOVF	Status stack or Loop stack overflow or PC stack full
SIG_TMZ0	Timer = 0 (high priority option)
SIG_VIRPTI	Vector Interrupt
SIG_IRQ2	Interrupt 2
SIG_IRQ1	Interrupt 1
SIG_IRQ0	Interrupt 0
SIG_SPR0I	DMA Channel 0 - SPORT0 Receive
SIG_SPR1I	DMA Channel 1 - SPORT1 Receive (or Link Buffer 0)
SIG_SPT0I	DMA Channel 2 - SPORT0 Transmit
SIG_SPT1I	DMA Channel 3 - SPORT1 Transmit (or Link Buffer 1)
[1]SIG_LP2I	DMA Channel 4 - Link Buffer 2
[1]SIG_LP3I	DMA Channel 5 - Link Buffer 3
SIG_EP0I	DMA Channel 6 - Ext. Port Buffer 0 (or Link Buffer 4)

Table 3-12. ADSP-2106x Processor Signals (Cont'd)

SIG Value	Definition
SIG_EP1I	DMA Channel 7 - Ext. Port Buffer 1 (or Link Buffer 5)
[1]SIG_EP2I	DMA Channel 8 - Ext. Port Buffer 2
[1]SIG_EP3I	DMA Channel 9 - Ext. Port Buffer 3
[1]SIG_LSRQ	Link port service request
SIG_CB7	Circular buffer 7 overflow
SIG_CB15	Circular buffer 15 overflow
SIG_TMZ	Timer = 0 (low priority option)
SIG_FIX	Fixed point overflow
SIG_FLTO	Floating point overflow exception
SIG_FLTU	Floating point underflow exception
SIG_FLTI	Floating point invalid exception
SIG_USR0	User software interrupt 0
SIG_USR1	User software interrupt 1
SIG_USR2	User software interrupt 2
SIG_USR3	User software interrupt 3

1 Signal is not present on the ADSP-21061 and ADSP-21065L processors.

Table 3-13. ADSP-2116x Processor Signals

SIG Value	Definition	Processor Restrictions
SIG_IICDI	Illegal input condition detected	
SIG_SOVF	Status stack or Loop stack overflow or PC stack full	
SIG_TMZ0	Timer = 0 (high priority option)	
SIG_VIRPTI	Vector interrupt	
SIG_IRQ2	Interrupt 2	

Table 3-13. ADSP-2116x Processor Signals (Cont'd)

SIG Value	Definition	Processor Restrictions
SIG_IRQ1	Interrupt 1	
SIG_IRQ0	Interrupt 0	
SIG_SPR0I	SPORT0 Receive	ADSP-21160 only
SIG_SPR1I	SPORT1 Receive	ADSP-21160 only
SIG_SPT01	SPORT0 Transmit	ADSP-21160 only
SIG_SPT1I	SPORT0 Transmit	ADSP-21160 only
SIG_SP0I	SPORT0 DMA	ADSP-21161 only
SIG_SP1I	SPORT1 DMA	ADSP-21161 only
SIG_SP2I	SPORT2 DMA	ADSP-21161 only
SIG_SP3I	SPORT3 DMA	ADSP-21161 only
SIG_LP0I	Link Buffer 0	
SIG_LP1I	Link Buffer 1	
SIG_LP2I	Link Buffer 2	ADSP-21160 only
SIG_LP3I	Link Buffer 3	ADSP-21160 only
SIG_LP4I	Link Buffer 4	ADSP-21160 only
SIG_LP5I	Link Buffer 5	ADSP-21160 only
SIG_SPIRI	SPI Receive DMA	ADSP-21161 only
SIG_SPITI	SPI Transmit DMA	ADSP-21161 only

Table 3-13. ADSP-2116x Processor Signals (Cont'd)

SIG Value	Definition	Processor Restrictions
SIG_EP0I	Ext. Port Buffer 0	
SIG_EP1I	Ext. Port Buffer 1	
SIG_EP2I	Ext. Port Buffer 2	
SIG_EP3I	Ext. Port Buffer 3	
SIG_LSRQ	Link port service request	
SIG_CB7	Circular buffer 7 overflow	
SIG_CB15	Circular buffer 15 overflow	
SIG_TMZ	Timer = 0 (low priority option)	
SIG_FIX	Fixed-point overflow	
SIG_FLTO	Floating-point overflow exception	
SIG_FLTU	Floating-point underflow exception	
SIG_FLTI	Floating-point invalid exception	
SIG_USR0	User software interrupt 0	
SIG_USR1	User software interrupt 1	

Table 3-14. ADSP-2126x Processor Signals

SIG Value	Definition
SIG_IICDI	Illegal input condition detected
SIG_SOVF	Status stack or Loop stack overflow or PC stack full
SIG_TMZ0	Timer = 0 (high priority option)
SIG_BKP	Hardware breakpoint
SIG_IRQ2	Interrupt 2
SIG_IRQ1	Interrupt 1
SIG_IRQ0	Interrupt 0

Table 3-14. ADSP-2126x Processor Signals (Cont'd)

SIG Value	Definition
SIG_DAIH	DAI High priority
SIG_SPIH	SPI transmit or receive (high priority option)
SIG_GPTMR0	General purpose IOP timer 0
SIG_SP1	SPORT 1
SIG_SP3	SPORT 3
SIG_SP5	SPORT 5 (ADSP-21262 and ADSP-21266 processors only)
SIG_SP0	SPORT 0
SIG_SP2	SPORT 2
SIG_SP4	SPORT 4 (ADSP-21262 and ADSP-21266 processors only)
SIG_PP	Parallel port
SIG_GPTMR1	General purpose IOP timer 1
SIG_DAIL	DAI low priority
SIG_GPTMR2	General purpose IOP timer 2
SIG_SPIL	SPI transmit or receive (low priority option)
SIG_CB7	Circular buffer 7 overflow
SIG_CB15	Circular buffer 15 overflow
SIG_TMZ	Timer = 0 (low priority option)
SIG_FIX	Fixed-point overflow
SIG_FLTO	Floating-point overflow exception
SIG_FLTU	Floating-point underflow exception
SIG_FLTI	Floating-point invalid exception
SIG_USR0	User software interrupt 0

Table 3-14. ADSP-2126x Processor Signals (Cont'd)

SIG Value	Definition
SIG_USR1	User software interrupt 1
SIG_USR2	User software interrupt 2
SIG_USR3	User software interrupt 3

Table 3-15. ADSP-2136x Processor Signals

SIG Value	Definition	Default setting (for programmable peripheral interrupts)
SIG_IICDI	Illegal input condition detected	
SIG_SOVF	Status stack or Loop stack overflow or PC stack full	
SIG_TMZ0	Timer = 0 (high priority option)	
SIG_BKP	Hardware breakpoint	
SIG_IRQ2	Interrupt 2	
SIG_IRQ1	Interrupt 1	
SIG_IRQ0	Interrupt 0	
SIG_P0	Peripheral interrupt - 0	DAI High priority
SIG_P1	Peripheral interrupt - 1	SPI transmit or receive (high priority option)
SIG_P2	Peripheral interrupt - 2	General purpose IOP timer 0
SIG_P3	Peripheral interrupt - 3	SPORT 1
SIG_P4	Peripheral interrupt - 4	SPORT 3
SIG_P5	Peripheral interrupt - 5	SPORT 5
SIG_P6	Peripheral interrupt - 6	SPORT 0
SIG_P7	Peripheral interrupt - 7	SPORT 2
SIG_P8	Peripheral interrupt - 8	SPORT 4

Table 3-15. ADSP-2136x Processor Signals (Cont'd)

SIG Value	Definition	Default setting (for programmable peripheral interrupts)
SIG_P9	Peripheral interrupt - 9	Parallel port
SIG_P10	Peripheral interrupt - 10	General purpose IOP timer 1
SIG_P12	Peripheral interrupt - 12	DAI low priority
SIG_P13	Peripheral interrupt - 13	PWM
SIG_P15	Peripheral interrupt - 15	DTCP
SIG_P17	Peripheral interrupt - 17	General purpose IOP timer 2
SIG_P18	Peripheral interrupt - 18	SPI transmit or receive (low priority option)
SIG_CB7	Circular buffer 7 overflow	
SIG_CB15	Circular buffer 15 overflow	
SIG_TMZ	Timer = 0 (low priority option)	
SIG_FIX	Fixed-point overflow	
SIG_FLTO	Floating-point overflow exception	
SIG_FLTU	Floating-point underflow exception	
SIG_FLTI	Floating-point invalid exception	
SIG_USR0	User software interrupt 0	
SIG_USR1	User software interrupt 1	
SIG_USR2	User software interrupt 2	
SIG_USR3	User software interrupt 3	

Error Conditions

The `clear_interrupt` function returns a 1 if the interrupt was pending; otherwise 0 is returned.

Example

```
#include <signal.h>
clear_interrupt (SIG_IRQ2);
   /* clear the interrupt 2 latch */
```

See Also

interrupt, raise, signal

clip

clip

Synopsis

```
#include <stdlib.h>
int clip (int value1, int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `clip` function returns its first argument if it is less than the absolute value of its second argument, otherwise it returns the absolute value of its second argument if the first is positive, or minus the absolute value if the first argument is negative. The `clip` function is a built-in function which is implemented with an `Rn = CLIP Rx BY Ry` instruction.

Error Conditions

The `clip` function does not return an error code.

Example

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

i = clip (10, 8);       /* returns 8  */
i = clip (8, 10);       /* returns 8  */
i = clip (-10, 8);      /* returns -8 */
```

See Also

fclip, lclip

clock

processor time

Synopsis

```
#include <time.h>
clock_t clock(void);
```

Description

The `clock` function returns the number of processor cycles that have elapsed since an arbitrary starting point. The function returns the value `(clock_t) -1`, if the processor time is not available or if it cannot be represented. The result returned by the function may be used to calculate the processor time in seconds by dividing it by the macro `CLOCKS_PER_SEC`. For more information, see "time.h" on page 3-27. An alternative method of measuring the performance of an application is described in "Measuring Cycle Counts" on page 3-40.

Error Conditions

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

Example

```
#include <time.h>

time_t start_time,stop_time;
double time_used;

start_time = clock();
compute();
stop_time = clock();

time_used = ((double) (stop_time - start_time)) / CLOCKS_PER_SEC;
```

See Also

No references to this function.

cos

cosine

Synopsis

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

Description

The cos functions return the cosine of the first argument. The input is interpreted as radians; the output is in the range [-1, 1].

Error Conditions

The input argument `x` for `cosf` must be in the domain [-1.647e6, 1.647e6] and the input argument for `cosd` must be in the domain [-8.433e8, 8.433e8]. The functions return zero if `x` is outside their domain.

Example

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

y = cos (3.14159);      /* y = -1.0 */
x = cosf (3.14159);     /* x = -1.0 */
```

See Also

acos, sin

cosh

hyperbolic cosine

Synopsis

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

Description

The cosh functions return the hyperbolic cosine of their argument.

Error Conditions

The domain of `coshf` is [-89.39, 89.39], and the domain for `coshd` is [-710.44, 710.44]. The functions return `HUGE_VAL` if the input argument `x` is outside the respective domains.

Example

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

y = cosh (x);
v = coshf (w);
```

See Also

sinh

count_ones

count one bits in word

Synopsis

```
#include <stdlib.h>
int count_ones (int value);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `count_ones` function returns the number of one bits in its argument.

Error Conditions

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

Example

```
#include <stdlib.h>
int flags1 = 0xAD1;
int flags2 = -1;
int cnt1;
int cnt2;

cnt1 = count_ones (flags1);     /* returns 6 */
cnt2 = count_ones (flags2);     /* returns 32 */
```

See Also

lcount_ones

ctime

convert calendar time into a string

Synopsis

```
#include <time.h>
char *ctime(const time_t *t);
```

Description

The `ctime` function converts a calendar time, pointed to by the argument `t` into a string that represents the local date and time. The form of the string is the same as that generated by `asctime`, and so a call to `ctime` is equivalent to

```
asctime(localtime(&t))
```

A pointer to the string is returned by `ctime`, and it may be overwritten by a subsequent call to the function.

Error Conditions

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

Example

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

time_t cal_time;

if (cal_time != (time_t)-1)
    printf("Date and Time is %s",ctime(&cal_time));
```

See Also

asctime, gmtime, localtime, time

difftime

difference between two calendar times

Synopsis

```
#include <time.h>
double difftime(time_t t1, time_t t0);
```

Description

The `difftime` function returns the difference in seconds between two calendar times, expressed as a `double`. By default, the `double` data type represents a 32-bit, single precision, floating-point, value. This form is normally insufficient to preserve all of the bits associated with the difference between two calendar times, particularly if the difference represents more than 97 days. It is recommended therefore that any function that calls `difftime` is compiled with the `-double-size-64` switch.

Error Conditions

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

Example

```
#include <time.h>
#include <stdio.h>
#define NA ((time_t)(-1))

time_t cal_time1;
time_t cal_time2;
double time_diff;

if ((cal_time1 == NA) || (cal_time2 == NA))
    printf("calendar time difference is not available\n");
else
    time_diff = difftime(cal_time2,cal_time1);
```

See Also

time

div

division

Synopsis

```
#include <stdlib.h>
div_t div (int numer, int denom);
```

Description

The `div` function divides `numer` by `denom`, both of type `int`, and returns a structure of type `div_t`. The type `div_t` is defined as

```
typedef struct {
   int quot;
   int rem;
} div_t;
```

where `quot` is the quotient of the division and `rem` is the remainder, such that if `result` is of type `div_t`, then

```
result.quot * denom + result.rem == numer
```

Error Conditions

If `denom` is zero, the behavior of the `div` function is undefined.

Example

```
#include <stdlib.h>
div_t result;

result = div (5, 2);    /* result.quot = 2, result.rem = 1 */
```

See Also

fmod, ldiv, modf

exit

normal program termination

Synopsis

```
#include <stdlib.h>
void exit (int status);
```

Description

The `exit` function causes normal program termination. The functions registered by the `atexit` function are called in reverse order of their registration and the microprocessor is put into the `IDLE` state. The `status` argument is stored in register `R0`, and control is passed to the label `___lib_prog_term`, which is defined in the run-time startup file.

Error Conditions

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

Example

```
#include <stdlib.h>

exit (EXIT_SUCCESS);
```

See Also

abort, atexit

exp

exponential

Synopsis

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

Description

The exp functions compute the exponential value `e` to the power of their argument.

Error Conditions

The input argument `x` for `expf` must be in the domain [-87.33, 88.72] and the input argument for `expd` must be in the domain [-708.2, 709.1]. The functions return `HUGE_VAL` if `x` is greater than the domain and 0.0 if `x` is less than the domain.

Example

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

y = exp (1.0);      /* y = 2.71828 */
x = expf (1.0);     /* x = 2.71828 */
```

See Also

log, pow

fabs

absolute value

Synopsis

```
#include <math.h>
double fabs (double x);
float fabsf (float x);
long double fabsd (long double x);
```

Description

The fabs functions return the absolute value of the argument `x`.

Error Conditions

The fabs functions do not return error conditions.

Example

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

y = fabs (-2.3);      /* y = 2.3 */
y = fabs (2.3);       /* y = 2.3 */
x = fabsf (-5.1);     /* x = 5.1 */
```

See Also

abs, labs

floor

floor

Synopsis

```
#include <math.h>
double floor (double x);
float floorf (float x);
long double floord (long double x);
```

Description

The floor functions return the largest integral value that is not greater than their argument.

Error Conditions

The floor functions do not return error conditions.

Example

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

y = floor (1.25);     /* y = 1.0  */
y = floor (-1.25);    /* y = -2.0 */
z = floorf (10.1);    /* z = 10.0 */
```

See Also

ceil

fmod

floating-point modulus

Synopsis

```
#include <math.h>
double fmod (double x, double y);
float fmodf (float x, float y);
long double fmodd (long double x, long double y);
```

Description

The fmod functions compute the floating-point remainder that results from dividing the first argument by the second argument.

The result is less than the second argument and has the same sign as the first argument. If the second argument is equal to zero, the fmod functions return zero.

Error Conditions

The fmod functions do not return an error condition.

Example

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

y = fmod (5.0, 2.0);      /* y = 1.0 */
x = fmodf (4.0, 2.0);     /* x = 0.0 */
```

See Also

div, ldiv, modf

free

deallocate memory

Synopsis

```
#include <stdlib.h>
void free (void *ptr);
```

Description

The `free` function deallocates a pointer previously allocated to a range of memory (by `calloc` or `malloc`) to the free memory heap. If the pointer was not previously allocated by `calloc`, `malloc`, `realloc`, `heap_calloc`, `heap_malloc`, or `heap_realloc`, the behavior is undefined.

The `free` function returns the allocated memory to the heap from which it was allocated.

Error Conditions

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

Example

```
#include <stdlib.h>
char *ptr;

ptr = malloc (10);      /* Allocate 10 words from heap */
free (ptr);             /* Return space to free heap   */
```

See Also

calloc, heap_calloc, heap_free, heap_lookup_name, heap_malloc, heap_realloc, malloc, realloc, set_alloc_type

frexp

separate fraction and exponent

Synopsis

```
#include <math.h>
double frexp (double x, int *expptr);
float frexpf (float x, int *expptr);
long double frexpd (long double x, int *expptr);
```

Description

The frexp functions separate a floating-point input into a normalized fraction and a (base 2) exponent. The functions return a fraction in the interval [½, 1), and store a power of 2 in the integer pointed to by the second argument. If the input is zero, then both the fraction and the exponent is set to zero.

Error Conditions

The frexp functions do not return an error condition.

Example

```
#include <math.h>
double y;
float x;
int exponent;

y = frexp (2.0, &exponent);      /* y = 0.5, exponent = 2  */
x = frexpf (4.0, &exponent);     /* x = 0.5, exponent = 3  */
```

See Also

modf

getenv

get string definition from operating system

Synopsis

```
#include <stdlib.h>
char *getenv (const char *name);
```

Description

The `getenv` function polls the operating system to see if a string is defined. There is no default operating system for the SHARC processors, so `getenv` always returns NULL.

Error Conditions

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

Example

```
#include <stdlib.h>
char *ptr;

ptr = getenv ("ADI_DSP");      /* ptr = NULL */
```

See Also

system

gmtime

convert calendar time into broken-down time as UTC

Synopsis

```
#include <time.h>
struct tm *gmtime(const time_t *t);
```

Description

The `gmtime` function converts a pointer to a calendar time into a broken-down time in terms of Coordinated Universal Time (UTC). A broken-down time is a structured variable, which is described in "time.h" on page 3-27.

The broken-down time is returned by `gmtime` as a pointer to static memory, which may be overwritten by a subsequent call to either `gmtime`, or to `localtime`.

Error Conditions

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

Example

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

time_t cal_time;
struct tm *tm_ptr;

cal_time = time(NULL);
if (cal_time != (time_t) -1) {
    tm_ptr = gmtime(&cal_time);
    printf("The year is %4d\n",1900 + (tm_ptr->tm_year));
}
```

See Also

localtime, mktime, time

heap_calloc

allocate and initialize memory in a heap

Synopsis

```
#include <stdlib.h>
void *heap_calloc(int heap_index, size_t nelem, size_t size);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `heap_calloc` function allocates from the heap identified by `heap_index`, an array containing `nelem` elements of `size`, and stores zeros in all bytes of the array. If successful, it returns a pointer to this array; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type whose size in bytes is not greater than `size`. The memory may be deallocated with the `free` or `heap_free` function.

For more information on creating multiple run-time heaps, see section "Using Multiple Heaps" on page 1-185.

Error Conditions

The `heap_calloc` function returns the null pointer if unable to allocate the requested memory.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
   char *buf;
   int index;
   /* Obtain the heap index for "seg_hp2" */
   index = heap_lookup_name("seg_hp2");
```

```
    if (index < 0) {
       printf("Heap with name seg_hp2 not found\n");
       return 1;
    }

    /* Allocate memory for 128 characters from seg_hp2 */
    buf = (char *)heap_calloc(index,128,sizeof(char));
    if (buf != 0) {
       printf("Allocated space from %p\n", buf);
       free(buf);   /* free can be used to release the memory */
    } else {
       printf("Unable to allocate from seg_hp2\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_free, heap_lookup_name, heap_malloc, heap_realloc, malloc, realloc, set_alloc_type

heap_free

return memory to a heap

Synopsis

```
#include <stdlib.h>
void heap_free(int heap_index, void *ptr);
```

Description

This function is an Analog Devices extension to the ANSI standard.

If `ptr` is not a null pointer, the `heap_free` function deallocates the object whose address is `ptr`; otherwise, it does nothing. The argument `heap_index` must be the index of the heap from which the object pointed to by `ptr` was originally allocated. If the object was not allocated from the specified heap, then the behavior is undefined.

The `heap_free` function is somewhat faster than `free`, but `free` must be used if the heap from which the object was allocated is not known with certainty.

For more information on creating multiple run-time heaps, see section "Using Multiple Heaps" on page 1-185.

Error Conditions

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

Example

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
   char *buf;
   int index;
```

```
    /* Obtain the heap index for "seg_hp2" */
    index = heap_lookup_name("seg_hp2");
    if (index < 0) {
       printf("Heap with name seg_hp2 not found\n");
       return 1;
    }

    /* Allocate memory for 128 characters from seg_hp2 */
    buf = (char *)heap_calloc(index,128,sizeof(char));
    if (buf != 0) {
       printf("Allocated space from %p\n", buf);
       heap_free(index, buf);      /* heap_free can be used */
                                   /* to release the memory */
    } else {
       printf("Unable to allocate from seg_hp2\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_lookup_name, heap_malloc, heap_realloc, malloc, realloc, set_alloc_type

heap_install

sets up a multiple heap at runtime

Synopsis

```
#include <stdlib.h>
int heap_install(void *base, size_t length, int userid,
                 int pmdm           /* -1 dm, 1 pm */);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `heap_install` function sets up a memory heap (`base`) with a size specified by `length` at runtime. The dynamic heap is identified by the `userid` and resides in either DM or PM memory as specified by the `pmdm` argument.

On successful initialization, `heap_install()` returns the heap index allocated for the newly installed heap. If the operation is unsuccessful, then `heap_install()` returns -1.

Once the dynamic heap is initialized, heap space can be claimed using the `heap_malloc` routine and associated heap management routines.

Note that the `heap_lookup_name` function does not work with a heap dynamically initialized by `heap_install()`. The `heap_lookup_name` function only works with statically initialized heaps.

Error Conditions

The `heap_install` function returns -1 if initialization was unsuccessful. This may be because there is not enough space available in the `__heaps` table, or if a heap with the specified `userid` already exists.

Examples

<< Linker Description File >>

```
MEMORY
{
       ..
       seg_runtime_dm { TYPE(DM RAM)
           START(0x0005b000) END(0x0005dfff) WIDTH(32) }
       ..
   }

PROCESSOR p0
{
       ..

   SECTIONS
   {
       ..

       seg_runtime_dm
       {
           _start_of_seg_runtime_dm = .;
       } > seg_runtime_dm
    }
}
```

<< C Source File >>

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

   #define DM_MEM     -1
   #define ADDR_DM    &__start_of_seg_runtime_dm

   int main()
   {
      int i;
      int index;
      int *x;
```

```
    index = heap_install(
            (void *)ADDR_DM, 100, 3, DM_MEM);
    if (index != -1)
       x = heap_malloc(3, 90);

    if (x) {
       for (i = 0; i < 90; i++)
           x[i] = i;
    }

    return 0;
}
```

See Also

heap_lookup_name, heap_malloc

heap_lookup_name

obtain primary heap identifier

Synopsis

```
#include <stdlib.h>
int heap_lookup_name(char * user_id);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `heap_lookup_name` function returns the primary heap identifier of the heap with user identifier `user_id`, if there is such a heap; otherwise, -1 is returned. The primary heap identifier is the index of the heap descriptor record in the heap descriptor table. The user identifier for a heap is determined by a field in the heap descriptor record. The default heap always has user identifier 0.

For more information on multiple run-time heaps, see section "Using Multiple Heaps" on page 1-185.

Error Conditions

The function returns -1 if the specified user identifier was not found, otherwise it returns the primary heap identifier of the specified heap.

Example

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

void func2(int pm * b);

func()
{
   int pm * x;
```

```
    int loop, pm_heapID;

    pm_heapID = heap_lookup_name("seg_heaq");

    if (pm_heapID < 0) {
       printf("Lookup failed\n");
       return 1;
    }

    x = (int pm *)heap_malloc(pm_heapID, 1000);
                              // Get 1K words of PM heap space
    if (x == NULL) {
       printf("heap_malloc failed\n");
       return 1;
    }

    for (loop = 0; loop < 1000; loop++)
        x[loop] = loop;

    func2(x);                 // Do something with x
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_malloc, heap_realloc, malloc, realloc, set_alloc_type

heap_malloc

allocate memory from a heap

Synopsis

```
#include <stdlib.h>
void *heap_malloc(int heap_index, size_t size);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `heap_malloc` function allocates an object of `size` from the heap identified by `heap_index`. It returns the address of the object if successful; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type whose size in bytes is not greater than `size`.

The block of memory is uninitialized. The memory may be deallocated with the `free` or `heap_free` function.

For more information on creating multiple run-time heaps, see section "Using Multiple Heaps" on page 1-185.

Error Conditions

The `heap_malloc` function returns the null pointer if unable to allocate the requested memory.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    char *buf;
    int index;
```

```
    /* Obtain the heap index for "seg_hp2" */
    index = heap_lookup_name("seg_hp2");
    if (index < 0) {
       printf("Heap with name seg_hp2 not found\n");
       return 1;
    }

    /* Allocate memory for 128 characters from seg_hp2 */
    buf = (char *)heap_malloc(index,128);
    if (buf != 0) {
       printf("Allocated space from %p\n", buf);
       free(buf); /* free can be used to release the memory */
    } else {
       printf("Unable to allocate from seg_hp2\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_lookup_name, heap_realloc, malloc, realloc, set_alloc_type

heap_realloc

change memory allocation from a heap

Synopsis

```
#include <stdlib.h>
void *heap_realloc(int heap_index, void *ptr, size_t size);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `heap_realloc` function allocates from the heap, identified by `heap_index`, an object of `size`, obtaining initial stored values from the object whose address is `ptr`. It returns the address of the object if successful; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type that is not greater than `size` in length.

If `ptr` is not a null pointer, it must be the address of an existing object that you first allocate by calling `calloc`, `malloc`, `realloc`, `heap_calloc`, `heap_malloc`, or `heap_realloc`. The heap identified by `heap_index` must be the same as the heap from which the object was originally allocated; if it is not the same, the behavior is undefined. If the existing object is not larger than the newly allocated object, `heap_realloc` copies the entire existing object to the initial part of the allocated object. (The values stored in the remainder of the object are indeterminate.)

Otherwise, the function copies only the initial part of the existing object that fits in the allocated object. If `heap_realloc` succeeds in allocating a new object, it deallocates the existing object. Otherwise, the existing object is left unchanged.

If `ptr` is a null pointer, the values stored in the newly created object are indeterminate.

If `size` is 0 and `ptr` is not a null pointer, the object pointed to by `ptr` is deallocated and a null pointer is returned. However, if the object was originally allocated from a different heap from the heap identified by `heap_index`, the behavior is undefined.

The `heap_realloc` function is somewhat faster than `realloc` for resizing or deallocating an existing object, but `realloc` must be used if the heap from which the object was originally allocated is not known with certainty.

The allocated memory may be deallocated with the `free` or `heap_free` function.

For more information on creating multiple run-time heaps, see section "Using Multiple Heaps" on page 1-185.

Error Conditions

The `heap_realloc` function returns the null pointer if unable to allocate the requested memory.

Example

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main()
{
   int index,ok,prev;
   char *buf,*upd;

/* Obtain the heap index for the user identifier 2 */
   index = heap_lookup_name("seg_hp2");
   if (index < 0) {
      printf("Heap with name seg_hp2 not found\n");
      return 1;
   }
```

```
    /* Allocate memory for 128 characters from seg_hp2          */
    buf = (char *)heap_malloc(index,128);
    if (buf != 0) {
        strcpy(buf,"hello");
                                /* Change allocated size to 256 */
        upd = (char *)heap_realloc(index,buf,256);
        if (upd != 0) {
            printf("reallocated string for %s\n",upd);
            heap_free(index,upd);          /* Return to seg_hp2 */
        } else {
            free(buf);       /* free can be used to release buf */
        }
    } else {
        printf("Unable to allocate from seg_hp2\n");
    }
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_lookup_name, heap_malloc, malloc, realloc, set_alloc_type

interrupt

define interrupt handling

Synopsis

```
#include <signal.h>
void (*interrupt (int sig, void(*func)(int))) (int);
void (*interruptnsm (int sig, void(*func)(int))) (int);
void (*interruptf (int sig, void(*func)(int))) (int);
void (*interruptfnsm (int sig, void(*func)(int))) (int);
void (*interrupts (int sig, void(*func)(int))) (int);
void (*interruptsnsm (int sig, void(*func)(int))) (int);
void (*interruptcb (int sig, void(*func)(int))) (int);
void (*interruptcbnsm (int sig, void(*func)(int))) (int);
void (*ininterruptss int sig, void(*func)(int))) (int);
void (*interruptssnsm int sig, void(*func)(int))) (int);
```

Description

The `interrupt` function determines how a signal received during program execution is handled. The `interrupt` function executes the function pointed to by `func` at every interrupt `sig`; the `signal` function executes the function only once. The `func` argument must be one of the following that are listed in Table 3-16. The `interrupt` function causes the receipt of the signal number `sig` to be handled in one of the following ways:

Table 3-16. 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 `func` is executed.

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

The differences between the functions `interrupt`, `interruptf`, `interrupts`, `interruptcb`, `interruptnsm`, `interruptfnsm`, `interruptsnsm`, `interruptcbnsm`, `interruptss`, and `interruptssnsm` are discussed under "signal.h" on page 3-18.

Error Conditions

The `interrupt` function returns `SIG_ERR` and sets `errno` equal to `SIG_ERR` if the requested interrupt is not recognized.

Example

```
#include <signal.h>

interrupt (SIG_IRQ2, irq2_handler);
/* enable interrupt 2 whose handling routine is pointed to by
irq2_handler */

interrupt (SIG_IRQ2, SIG_IGN);
/* disable interrupt 2 */
```

See Also

raise, signal

isalnum

detect alphanumeric character

Synopsis

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

Description

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

Error Conditions

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

Example

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

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

See Also

isalpha, isdigit

isalpha

detect alphabetic character

Synopsis

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

Description

The `isalpha` function determines if the argument is an alphabetic character (`A-Z` or `a-z`). If the argument is not alphabetic, `isalpha` returns a zero. If the argument is alphabetic, `isalpha` returns a nonzero value.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isalpha (ch) ? "alphabetic" : "");
    putchar ('\n');
}
```

See Also

isalnum, isdigit

iscntrl

detect control character

Synopsis

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

Description

The `iscntrl` function determines if the argument is a control character (`0x00-0x1F` or `0x7F`). If the argument is not a control character, `iscntrl` returns a zero. If the argument is a control character, `iscntrl` returns a non-zero value.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", iscntrl (ch) ? "control" : "");
    putchar ('\n');
}
```

See Also

isalnum, isgraph

isdigit

detect decimal digit

Synopsis

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

Description

The `isdigit` function determines if the argument `c` is a decimal digit (`0-9`). If the argument is not a digit, `isdigit` returns a zero. If the argument is a digit, `isdigit` returns a non-zero value.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isdigit (ch) ? "digit" : "");
    putchar ('\n');
}
```

See Also

isalnum, isalpha, isxdigit

isgraph

detect printable character, not including white space

Synopsis

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

Description

The `isgraph` function determines if the argument is a printable character, not including a white space (`0x21-0x7e`). If the argument is not a printable character, `isgraph` returns a zero. If the argument is a printable character, `isgraph` returns a non-zero value.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isgraph (ch) ? "graph" : "");
    putchar ('\n');
}
```

See Also

isalnum, iscntrl, isprint

isinf

test for infinity

Synopsis

```
#include <math.h>
int isinff(float x);
int isinf(double x);
int isinfd(long double x);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The isinf functions return a zero if the argument `x` is not set to the IEEE constant for `+Infinity` or `-Infinity`; otherwise, the functions return a non-zero value.

Error Conditions

The isinf functions do not return or set any error conditions.

Example

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

static int fail=0;

main(){
   /* test int isinf(double) */
   union {
      double d; float f; unsigned long l;
   } u;

   #ifdef __DOUBLES_ARE_FLOATS__
      u.l=0xFF800000L; if ( isinf(u.d)==0 ) fail++;
      u.l=0xFF800001L; if ( isinf(u.d)!=0 ) fail++;
```

```
        u.l=0x7F800000L; if ( isinf(u.d)==0 ) fail++;
        u.l=0x7F800001L; if ( isinf(u.d)!=0 ) fail++;
    #endif

    /* test int isinff(float) */
        u.l=0xFF800000L; if ( isinff(u.f)==0 ) fail++;
        u.l=0xFF800001L; if ( isinff(u.f)!=0 ) fail++;
        u.l=0x7F800000L; if ( isinff(u.f)==0 ) fail++;
        u.l=0x7F800001L; if ( isinff(u.f)!=0 ) fail++;

    /* print pass/fail message */
    if ( fail==0 )
        printf("Test passed\n");
    else
        printf("Test failed: %d\n", fail);
}
```

See Also

isnan

islower

detect lowercase character

Synopsis

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

Description

The `islower` function determines if the argument is a lowercase character (`a-z`). If the argument is not lowercase, `islower` returns a zero. If the argument is lowercase, `islower` returns a non-zero value.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", islower (ch) ? "lowercase" : "");
    putchar ('\n');
}
```

See Also

isalpha, isupper

isnan

test for Not a Number (NaN)

Synopsis

```
#include <math.h>
int isnanf(float x);
int isnan(double x);
int isnand(long double x);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The isnan functions return a zero if the argument `x` is not set to an IEEE `NaN` (Not a Number); otherwise, the functions return a non-zero value.

Error Conditions

The isnan functions do not return or set any error conditions.

Example

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

static int fail=0;

main(){
   /* test int isnan(double) */
   union {
      double d; float f; unsigned long l;
   } u;

   #ifdef __DOUBLES_ARE_FLOATS__
      u.l=0xFF800000L; if ( isnan(u.d)!=0 ) fail++;
      u.l=0xFF800001L; if ( isnan(u.d)==0 ) fail++;
      u.l=0x7F800000L; if ( isnan(u.d)!=0 ) fail++;
```

```
        u.l=0x7F800001L; if ( isnan(u.d)==0 ) fail++;
    #endif

    /* test int isnanf(float) */
        u.l=0xFF800000L; if ( isnanf(u.f)!=0 ) fail++;
        u.l=0xFF800001L; if ( isnanf(u.f)==0 ) fail++;
        u.l=0x7F800000L; if ( isnanf(u.f)!=0 ) fail++;
        u.l=0x7F800001L; if ( isnanf(u.f)==0 ) fail++;

    /* print pass/fail message */
    if ( fail==0 )
        printf("Test passed\n");
    else
        printf("Test failed: %d\n", fail);
}
```

See Also

isinf

isprint

detect printable character

Synopsis

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

Description

The `isprint` function determines if the argument is a printable character (`0x20-0x7E`). If the argument is not a printable character, `isprint` returns a zero. If the argument is a printable character, `isprint` returns a non-zero value.

Error Conditions

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

Example

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

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

See Also

isgraph, isspace

ispunct

detect punctuation character

Synopsis

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

Description

The `ispunct` function determines if the argument is a punctuation character. If the argument is not a punctuation character, `ispunct` returns a zero. If the argument is a punctuation character, `ispunct` returns a non-zero value.

Error Conditions

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

Example

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

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

See Also

isalnum

isspace

detect whitespace character

Synopsis

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

Description

The `isspace` function determines if the argument is a blank whitespace character (`0x09-0x0D` or `0x20`). This includes space `( )`, form feed (`\f`), new line (`\n`), carriage return (`\r`), horizontal tab (`\t`) and vertical tab (`\v`).

If the argument is not a blank whitespace character, `isspace` returns a zero. If the argument is a blank whitespace character, `isspace` returns a non-zero value.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isspace (ch) ? "space" : "");
    putchar ('\n');
}
```

See Also

iscntrl, isgraph

isupper

detect uppercase character

Synopsis

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

Description

The `isupper` function determines if the argument is an uppercase character (`A-Z`). If the argument is not an uppercase character, `isupper` returns a zero. If the argument is an uppercase character, `isupper` returns a non-zero value.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isupper (ch) ? "uppercase" : "");
    putchar ('\n');
}
```

See Also

isalpha, islower

isxdigit

detect hexadecimal digit

Synopsis

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

Description

The `isxdigit` function determines if the argument is a hexadecimal digit character (`A-F`, `a-f`, or `0-9`). If the argument is not a hexadecimal digit, `isxdigit` returns a zero. If the argument is a hexadecimal digit, `isxdigit` returns a non-zero value.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    printf ("%2s", isxdigit (ch) ? "hexadecimal" : "");
    putchar ('\n');
}
```

See Also

isalnum, isdigit

labs

absolute value

Synopsis

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

Description

The `labs` function returns the absolute value of its integer argument.

Note that `labs (LONG_MIN) == LONG_MIN`.

Error Conditions

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

Example

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

j = labs (-285128);      /* j = 285128 */
```

See Also

abs, fabs

lavg

mean of two values

Synopsis

```
#include <stdlib.h>
long int lavg (long int value1, long int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `lavg` function adds two arguments and divides the result by two. The `lavg` function is a built-in function which is implemented with an `Rn=(Rx+Ry)/2` instruction.

Error Conditions

The `lavg` function does not return an error code.

Example

```
#include <stdlib.h>
long int i;
i = lavg (10, 8);          /* returns 9 */
```

See Also

avg, favg

lclip

clip

Synopsis

```
#include <stdlib.h>
long int lclip (long int value1, long int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `lclip` function returns its first argument if it is less than the absolute value of its second argument, otherwise it returns the absolute value of its second argument if the first is positive, or minus the absolute value if the first argument is negative. The `lclip` function is a built-in function which is implemented with an `Rn = CLIP Rx BY Ry` instruction.

Error Conditions

The `lclip` function does not return an error code.

Example

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

i = lclip (10, 8);      /* returns 8 */
i = lclip (8, 10);      /* returns 8 */
i = lclip (-10, 8);     /* returns -8 */
```

See Also

clip, fclip

lcount_ones

count one bits in word

Synopsis

```
#include <stdlib.h>
int lcount_ones (long int value);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `lcount_ones` function returns the number of one bits in its argument.

Error Conditions

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

Example

```
#include <stdlib.h>
long int flags1 = 4095;
long int flags2 = 4096;
int cnt1;
int cnt2;

cnt1 = lcount_ones (flags1);    /* returns 12 */
cnt2 = lcount_ones (flags2);    /* returns 1 */
```

See Also

count_ones

ldexp

multiply by power of 2

Synopsis

```
#include <math.h>
double ldexp (double x, int n);
float ldexpf (float x, int n);
long double ldexpd (long double x, int n);
```

Description

The ldexp functions return the value of the floating-point argument multiplied by 2^n. These functions add the value of `n` to the exponent of `x`.

Error Conditions

If the result overflows, the ldexp functions return `HUGE_VAL` with the proper sign and set `errno` to `ERANGE`. If the result underflows, a zero is returned.

Example

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

y = ldexp (0.5, 2);       /* y = 2.0  */
x = ldexpf (1.0, 2);      /* x = 4.0  */
```

See Also

exp, pow

ldiv

long division

Synopsis

```
#include <stdlib.h>
ldiv_t ldiv (long int numer, long int denom);
```

Description

The `ldiv` function divides `numer` by `denom`, and returns a structure of type `ldiv_t`. The type `ldiv_t` is defined as:

```
typedef struct {
    long int quot;
    long int rem;
} ldiv_t;
```

where `quot` is the quotient of the division and `rem` is the remainder, such that if result is of type `ldiv_t`, then

```
result.quot * denom + result.rem == numer
```

Error Conditions

If `denom` is zero, the behavior of the `ldiv` function is undefined.

Example

```
#include <stdlib.h>
ldiv_t result;

result = ldiv (7L, 2L);    /* result.quot = 3, result.rem = 1 */
```

See Also

div, fmod

lmax

long maximum

Synopsis

```
#include <stdlib.h>
long int lmax (long int value1, long int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `lmax` function returns the larger of its two arguments. The `lmax` function is a built-in function which is implemented with an `Rn = MAX(Rx,Ry)` instruction.

Error Conditions

The `lmax` function does not return an error code.

Example

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

i = lmax (10L, 8L);     /* returns 10 */
```

See Also

fmax, fmin, lmin, max, min

lmin

long minimum

Synopsis

```
#include <stdlib.h>
long int lmin (long int value1, long int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `lmin` function returns the smaller of its two arguments. The `lmin` function is a built-in function which is implemented with an `Rn = MIN(Rx,Ry)` instruction.

Error Conditions

The `lmin` function does not return an error code.

Example

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

i = lmin (10L, 8L);     /* returns 8 */
```

See Also

fmax, fmin, lmax, max, min

localeconv

get pointer for formatting to current locale

Synopsis

```
#include <locale.h>
struct lconv *localeconv (void);
```

Description

The `localeconv` function returns a pointer to an object of type `struct lconv`. This pointer is used to set the components of the object with values used in formatting numeric quantities in the current locale.

With the exception of `decimal_point`, those members of the structure with type `char*` may use " " to indicate that a value is not available. Expected values are strings. Those members with type `char` may use `CHAR_MAX` to indicate that a value is not available. Expected values are non-negative numbers.

The program may not alter the structure pointed to by the return value but subsequent calls to `localeconv` may do so. Also, calls to `setlocale` with the category arguments of `LC_ALL`, `LC_MONETARY` and `LC_NUMERIC` may overwrite the structure.

Table 3-17. Members of the lconv Struct

Member	Description
`char *currency_symbol`	Currency symbol applicable to the locale
`char *decimal_point`	Used to format nonmonetary quantities
`char *grouping`	Used to indicate the number of digits in each nonmonetary grouping
`char *int_curr_symbol`	Used as international currency symbol (ISO 4217:1987) for that particular locale plus the symbol used to separate the currency symbol from the monetary quantity

Table 3-17. Members of the lconv Struct (Cont'd)

Member	Description
`char *mon_decimal_point`	Used for decimal point format monetary quantities
`char *mon_grouping`	Used to indicate the number of digits in each monetary grouping
`char *mon_thousands_sep`	Used to group monetary quantities prior to the decimal point
`char *negative_sign`	Used to indicate a negative monetary quantity
`char *positive_sign`	Used to indicate a positive monetary quantity
`char *thousands_sep`	Used to group nonmonetary quantities prior to the decimal point
`char frac_digits`	Number of digits displayed after the decimal point in monetary quantities in other than international format
`char int_frac_digits`	Number of digits displayed after the decimal point in international monetary quantities
`char p_cs_precedes`	If set to 1, the `currency_symbol` precedes the positive monetary quantity. If set to 0, the `currency_symbol` succeeds the positive monetary quantity.
`char n_cs_precedes`	If set to 1, the `currency_symbol` precedes the negative monetary quantity. If set to 0, the `currency_symbol` succeeds the negative monetary quantity.
`char n_sign_posn`	Indicates the positioning of `negative_sign` for monetary quantities.
`char n_sep_by_space`	If set to 1, the `currency_symbol` is separated from the negative monetary quantity. If set to 0, the `currency_symbol` is not separated from the negative monetary quantity.
`char p_sep_by_space`	If set to 1, the `currency_symbol` is separated from the positive monetary quantity. If set to 0, the `currency_symbol` is not separated from the positive monetary quantity.

For grouping and non_grouping, an element of CHAR_MAX indicates that no further grouping will be performed, a 0 indicates that the previous element should be used to group the remaining digits, and any other integer value is used as the number of digits in the current grouping.

The definitions of the values for p_sign_posn and n_sign_posn are:

- parentheses surround currency_symbol and quantity
- sign string precedes currency_symbol and quantity
- sign string succeeds currency_symbol and quantity
- sign string immediately precedes currency_symbol
- sign string immediately succeeds currency_symbol

Error Conditions

The localeconv function does not return an error condition.

Example

```
#include <locale.h>
struct lconv *c_locale;

c_locale = localeconv ();     /* Only the C locale is */
                              /* currently supported  */
```

See Also

setlocale

localtime

convert calendar time into broken-down time

Synopsis

```
#include <time.h>
struct tm *localtime(const time_t *t);
```

Description

The `localtime` function converts a pointer to a calendar time into a broken-down time that corresponds to current time zone. A broken-down time is a structured variable, which is described in "time.h" on page 3-27. This implementation of the header file does not support the Daylight Saving flag nor does it support time zones and, thus, `localtime` is equivalent to the `gmtime` function.

The broken-down time is returned by `localtime` as a pointer to static memory, which may be overwritten by a subsequent call to either `localtime`, or to `gmtime`.

Error Conditions

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

Example

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

time_t cal_time;
struct tm *tm_ptr;

cal_time = time(NULL);
if (cal_time != (time_t) -1) {
    tm_ptr = localtime(&cal_time);
    printf("The year is %4d\n",1900 + (tm_ptr->tm_year));
}
```

See Also

asctime, gmtime, mktime, time

log

natural logarithm

Synopsis

```
#include <math.h>
double log (double x);
float logf (float x);
long double logd (long double x);
```

Description

The log functions compute the natural (base e) logarithm of their argument.

Error Conditions

The log functions return zero and set `errno` to `EDOM` if the input value is zero or negative.

Example

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

y = log (1.0);          /* y = 0.0 */
x = logf (2.71828);     /* x = 1.0 */
```

See Also

alog, exp, log10

log10

base 10 logarithm

Synopsis

```
#include <math.h>
double log10 (double x);
float log10f (float x);
long double log10d (long double x);
```

Description

The log10 functions produce the base 10 logarithm of their argument.

Error Conditions

The log10 functions indicate a domain error (set `errno` to `EDOM`) and return zero if the input is zero or negative.

Example

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

y = log10 (100.0);     /* y = 2.0 */
x = log10f (10.0);     /* x = 1.0 */
```

See Also

alog10, log, pow

longjmp

second return from `setjmp`

Synopsis

```
#include <setjmp.h>
void longjmp (jmp_buf env, int return_val);
```

Description

The `longjmp` function causes the program to execute a second return from the place where `setjmp (env)` was called (with the same `jmp_buf` argument).

The `longjmp` function takes as its arguments a jump buffer that contains the context at the time of the original call to `setjmp`. It also takes an integer, `return_val`, which `setjmp` returns if `return_val` is non-zero. Otherwise, `setjmp` returns a 1.

If `env` was not initialized through a previous call to `setjmp` or the function that called `setjmp` has since returned, the behavior is undefined. Also, automatic variables that are local to the original function calling `setjmp`, that do not have `volatile`-qualified type, and that have changed their value prior to the `longjmp` call, have indeterminate value.

Error Conditions

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

Example

```
#include <setjmp.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
jmp_buf env;
int res;
```

```
if ((res == setjmp(env)) != 0) {
    printf  ("Problem %d reported by func ()", res);
    exit (EXIT_FAILURE);
}
func ();

void func (void)
{
    if (errno != 0) {
        longjmp (env, errno);
    }
}
```

See Also

setjmp

malloc

allocate memory

Synopsis

```
#include <stdlib.h>
void *malloc (size_t size);
```

Description

The `malloc` function returns a pointer to a block of memory of length `size`. The block of memory is uninitialized.

The object is allocated from the current heap, which is the default heap unless `set_alloc_type` has been called to change the current heap to an alternate heap.

Error Conditions

The `malloc` function returns a null pointer if it is unable to allocate the requested memory.

Example

```
#include <stdlib.h>
int *ptr;

ptr = (int *)malloc (10);     /* ptr points to an   */
                              /* array of length 10 */
```

See Also

calloc, free, heap_calloc, heap_free, heap_lookup_name, heap_malloc, heap_realloc, realloc, set_alloc_type

max

maximum

Synopsis

```
#include <stdlib.h>
int max (int value1, int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `max` function returns the larger of its two arguments. The `max` function is a built-in function which is implemented with an `Rn = MAX(Rx,Ry)` instruction.

Error Conditions

The `max` function does not return an error code.

Example

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

i = max (10, 8);     /* returns 10 */
```

See Also

fmax, fmin, lmax, lmin, min

memchr

find first occurrence of character

Synopsis

```
#include <string.h>
void *memchr (const void *s1, int c, size_t n);
```

Description

The `memchr` function compares the range of memory pointed to by `s1` with the input character `c` and returns a pointer to the first occurrence of `c`. A null pointer is returned if `c` does not occur in the first `n` characters.

Error Conditions

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

Example

```
#include <string.h>
char *ptr;

ptr = memchr ("TESTING", 'E', 7);
    /* ptr points to the E in TESTING */
```

See Also

strchr, strrchr

memcmp

compare objects

Synopsis

```
#include <string.h>
int memcmp (const void *s1, const void *s2, size_t n);
```

Description

The `memcmp` function compares the first `n` characters of the objects pointed to by `s1` and `s2`. It returns a positive value if the `s1` object is lexically greater than the `s2` object, a negative value if the `s2` object is lexically greater than the `s1` object, and a zero if the objects are the same.

Error Conditions

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

Example

```
#include <string.h>
char string1 = "ABC";
char string2 = "BCD";
int result;

result = memcmp (string1, string2, 3);    /* result < 0 */
```

See Also

strcmp, strcoll, strncmp

memcpy

copy characters from one object to another

Synopsis

```
#include <string.h>
void *memcpy (void *s1, const void *s2, size_t n);
```

Description

The `memcpy` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The behavior of `memcpy` is undefined if the two objects overlap. For more information, see "memmove" on page 3-171.

The `memcpy` function returns the address of `s1`.

Error Conditions

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

Example

```
#include <string.h>
char *a = "SRC";
char *b = "DEST";

memcpy (b, a, 3);    /* b = "SRCT" */
```

See Also

memmove, strcpy, strncpy

memmove

copy characters between overlapping objects

Synopsis

```
#include <string.h>
void *memmove (void *s1, const void *s2, size_t n);
```

Description

The `memmove` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The entire object is copied correctly even if the objects overlap.

The `memmove` function returns a pointer to `s1`.

Error Conditions

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

Example

```
#include <string.h>
char *ptr, *str = "ABCDE";

ptr = str + 2;
memmove (ptr, str, 3);    /* ptr = "ABC", str = "ABABC" */
```

See Also

memcpy, strcpy, strncpy

memset

set range of memory to a character

Synopsis

```
#include <string.h>
void *memset (void *s1, int c, size_t n);
```

Description

The `memset` function sets a range of memory to the input character `c`. The first `n` characters of `s1` are set to `c`.

The `memset` function returns a pointer to `s1`.

Error Conditions

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

Example

```
#include <string.h>
char string1[50];

memset (string1, '\0', 50);    /* set string1 to 0 */
```

See Also

memcpy

min

minimum

Synopsis

```
#include <stdlib.h>
int min (int value1, int value2);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `min` function returns the smaller of its two arguments. The `min` function is a built-in function which is implemented with an `Rn=MIN(Rx,Ry)` instruction.

Error Conditions

The `min` function does not return an error code.

Example

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

i = min (10, 8);     /* returns 8 */
```

See Also

fmax, fmin, lmax, lmin, max

mktime

convert broken-down time into a calendar time

Synopsis

```
#include <time.h>
time_t mktime(struct tm *tm_ptr);
```

Description

The `mktime` function converts a pointer to a broken-down time, which represents a local date and time, into a calendar time. However, this implementation of `time.h` does not support either daylight saving or time zones and hence this function will interpret the argument as Greenwich Mean Time (UTC).

A broken-down time is a structured variable which is defined in the `time.h` header file as:

```
struct tm { int tm_sec;   /* seconds after the minute [0,61] */
   int tm_min;            /* minutes after the hour [0,59]   */
   int tm_hour;           /* hours after midnight [0,23]     */
   int tm_mday;           /* day of the month [1,31]         */
   int tm_mon;            /* months since January [0,11]     */
   int tm_year;           /* years since 1900                */
   int tm_wday;           /* days since Sunday [0, 6]        */
   int tm_yday;           /* days since January 1st [0,365]  */
   int tm_isdst;          /* Daylight Saving flag            */
};
```

The various components of the broken-down time are not restricted to the ranges indicated above. The `mktime` function calculates the calendar time from the specified values of the components (ignoring the initial values of `tm_wday` and `tm_yday`), and then"'normalizes" the broken-down time forcing each component into its defined range.

Error Conditions

The `mktime` function returns the value `(time_t) -1` if the calendar time cannot be represented.

Example

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

static const char *wday[] = {"Sun","Mon","Tue","Wed",
                             "Thu","Fri","Sat","???"};

struct tm tm_time = {0,0,0,0,0,0,0,0,0};

tm_time.tm_year = 2000 - 1900;
tm_time.tm_mday = 1;

if (mktime(&tm_time) == -1)
    tm_time.tm_wday = 7;
printf("%4d started on a %s\n",
    1900 + tm_time.tm_year,
    wday[tm_time.tm_wday]);
```

See Also

gmtime, localtime, time

modf

separate integral and fractional parts

Synopsis

```
#include <math.h>
double modf (double x, double *intptr);
float modff (float x, float *intptr);
long double modfd (long double x, long double *intptr);
```

Description

The modf functions separate the first argument into integral and fractional portions. The fractional portion is returned and the integral portion is stored in the object pointed to by `intptr`. The integral and fractional portions have the same sign as the input.

Error Conditions

The modf functions do not return error conditions.

Example

```
#include <math.h>
double y, n;
float m, p;

y = modf (-12.345, &n);     /* y = -0.345, n = -12.0 */
m = modff (11.75, &p);      /* m = 0.75, p = 11.0    */
```

See Also

frexp

pow

raise to a power

Synopsis

```
#include <math.h>
double pow (double x, double y);
float powf (float x, float y);
long double powd (long double x, long double y);
```

Description

The pow functions compute the value of the first argument raised to the power of the second argument.

Error Conditions

A domain error occurs if the first argument is negative and the second argument cannot be represented as an integer. If the first argument is zero, the second argument is less than or equal to zero and the result cannot be represented, `EDOM` is stored in `errno` and zero is returned.

Example

```
#include <math.h>
double z;
float x;

z = pow (4.0, 2.0);      /* z = 16.0 */
x = powf (4.0, 2.0);     /* x = 16.0 */
```

See Also

exp, ldexp

qsort

quicksort

Synopsis

```
#include <stdlib.h>
void qsort (void *base, size_t nelem, size_t size,
            int (*compar) (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 `compar`, 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, where

- `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.
- `compar` is a pointer to a function that is called by `qsort` to compare two elements of the array. The function should return a value less than, equal to, or greater than zero, according to whether the first argument is less than, equal to, or greater than the second.

Error Conditions

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

Example

```
#include <stdlib.h>
float a[10];

int compare_float (const void *a, const void *b)
{
    float aval = *(float *)a;
    float bval = *(float *)b;
    if (aval < bval)
        return -1;
    else if (aval = = bval)
        return 0;
    else
        return 1;
}

qsort (a, sizeof (a)/sizeof (a[0]), sizeof (a[0]), compare_float);
```

See Also

bsearch

raise

force a signal

Synopsis

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

Description

This function is an Analog Devices extension to the ANSI standard.

The `raise` function sends the signal `sig` to the executing program. The `raise` function forces interrupts wherever possible and simulates an interrupt otherwise. The `sig` argument must be one of the signals listed in Table 3-11 on page 3-92, Table 3-12 on page 3-93, Table 3-13 on page 3-94, Table 3-14 on page 3-96, and Table 3-15 on page 3-98.

The `raise` function uses self-modifying code. If this is not suitable for your application, then use the `raisensm` function instead. The choice of function has no effect on the dispatcher used and no effect on the overall interrupt handling performance.

Error Conditions

The `raise` function returns a zero if successful or a nonzero value if it fails.

Example

```
#include <signal.h>
raise (SIG_IRQ2);    /* invoke the interrupt 2 handler */
```

See Also

interrupt, signal

rand

random number generator

Synopsis

```
#include <stdlib.h>
int rand (void);
```

Description

The `rand` function returns a pseudo-random integer value in the range $[0, 2^{31} - 1]$.

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 $2^{31} - 1$.

Error Conditions

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

Example

```
#include <stdlib.h>
int i;
i = rand ();
```

See Also

srand

realloc

change memory allocation

Synopsis

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

Description

The `realloc` function changes the memory allocation of the object pointed to by `ptr` to `size`. Initial values for the new object are taken from those in the object pointed to by `ptr`. If the size of the new object is greater than the size of the object pointed to by `ptr`, then the values in the newly allocated section are undefined. If `ptr` is a non-null pointer that was not allocated with `malloc` or `calloc`, the behavior is undefined. If `ptr` is a null pointer, `realloc` imitates `malloc`. If `size` is zero and `ptr` is not a null pointer, `realloc` imitates `free`.

If `ptr` is not a null pointer, then the object is re-allocated from the heap that the object was originally allocated from. If `ptr` is a null pointer, then the object is allocated from the current heap, which is the default heap unless `set_alloc_type` has been called to change the current heap to an alternate heap.

Error Conditions

If memory cannot be allocated, `ptr` remains unchanged and `realloc` returns a null pointer.

Example

```
#include <stdlib.h>
int *ptr;
```

```
ptr = (int *)malloc (10);          /* intervening code   */
ptr = (int *)realloc (ptr, 20);    /* the size is now 20 */
```

See Also

calloc, free, heap_calloc, heap_free, heap_lookup_name, heap_malloc, heap_realloc, malloc, set_alloc_type

setjmp

define a run-time label

Synopsis

```
#include <setjmp.h>
int setjmp (jmp_buf env);
```

Description

The `setjmp` function saves the calling environment in the `jmp_buf` argument. The effect of the call is to declare a run-time label that can be jumped to via a subsequent call to `longjmp`.

When `setjmp` is called, it immediately returns with a result of zero to indicate that the environment has been saved in the `jmp_buf` argument. If, at some later point, `longjmp` is called with the same `jmp_buf` argument, `longjmp` restores the environment from the argument. The execution is then resumed at the statement immediately following the corresponding call to `setjmp`. The effect is as if the call to `setjmp` has returned for a second time but this time the function returns a non-zero result.

The effect of calling `longjmp` is undefined if the function that called `setjmp` has returned in the interim.

Error Conditions

The label `setjmp` does not return an error condition.

Example

See “longjmp” on page 3-164

See Also

longjmp

setlocale

set the current locale

Synopsis

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

Description

The `setlocale` function uses the parameters `category` and `locale` to select a current locale. The possible values for the `category` argument are those macros defined in `locale.h` beginning with "`LC_`". The only `locale` argument supported at this time is the "`C`" locale. If a null pointer is used for the `locale` argument, `setlocale` returns a pointer to a string which is the current locale for the given category argument. A subsequent call to `setlocale` with the same `category` argument and the string supplied by the previous `setlocale` call returns the locale to its original status. The string pointed to may not be altered by the program but may be overwritten by subsequent `setlocale` calls.

Error Conditions

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

Example

```
#include <locale.h>

setlocale (LC_ALL, "C");
    /* sets the locale to the "C" locale */
```

See Also

localeconv

set_alloc_type

set heap for dynamic memory allocation

Synopsis

```
#include <stdlib.h>
int set_alloc_type(char * heap_name);
```

Description

This function is an Analog Devices extension to the ANSI standard.

The `set_alloc_type` function specifies a heap from which `malloc` and `calloc` should subsequently allocate memory. The `heap_name` argument should be the name of the segment containing the heap as a string. For more information on creating multiple heaps, see "Using Multiple Heaps" on page 1-185.

The `set_alloc_type` function is not available in multithreaded environments.

Error Conditions

The `set_alloc_type` function returns a non-zero value if the heap specified cannot be found.

Example

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

char *mymem, *stdmem;

int allocate()
{
    int res;
    res = set_alloc_type("seg_heaq");
```

```
    if (res != 0) {
        printf("Failed to switch heaps\n");
        return 1;
    }

    mymem = malloc(10);        /* mymem is allocated on "seg_heaq" */
    if (mymem == NULL) {
        printf("Failed to allocate memory from seg_heaq\n");
        return 1;
    }

    res = set_alloc_type("seg_heap");
    if (res != 0) {
        printf("Failed to switch heaps\n");
        return 1;
    }

    stdmem = malloc(10);  /* stdmem is allocated on the default heap */
    if (stdmem == NULL) {
        printf("Failed to allocate memory from the default heap\n");
        return 1;
    }

    printf("Memory was allocated at %p %p\n", mymem, stdmem);
    return 0;
}
```

See Also

calloc, free, heap_calloc, heap_free, heap_lookup_name, heap_malloc, heap_realloc, malloc, realloc

signal

define signal handling

Synopsis

```
#include <signal.h>
void (*signal (int sig, void (*func)(int))) (int);
void (*signalnsm (int sig, void (*func)(int))) (int);
void (*signalf (int sig, void (*func)(int))) (int);
void (*signalfnsm (int sig, void (*func)(int))) (int);
void (*signals (int sig, void (*func)(int))) (int);
void (*signalsnsm (int sig, void (*func)(int))) (int);
void (*signalcb (int sig, void (*func)(int))) (int);
void (*signalcbnsm (int sig, void (*func)(int))) (int);
void (*signalss  (int sig, void (*func)(int))) (int);
void (*signalssnsm (int sig, void (*func)(int))) (int);
```

Description

The `signal`, `signalnsm`, `signalf`, `signalfnsm`, `signals`, `signalsnsm`, `signalcb`, `signalcbnsm`, `signalss` or `signalssnsm` functions determine how a signal that is received during program execution is handled. The specified `signal` function causes the corresponding `interrupt` dispatcher to be used when handling the interrupt (refer to "signal.h" on page 3-18 for more information).

The `signal` function 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 either Table 3-11 on page 3-92, Table 3-12 on page 3-93, Table 3-13 on page 3-94, Table 3-14 on page 3-96, or Table 3-15 on page 3-98. The `signal` function causes the receipt of the signal number `sig` to be handled in one of the ways listed in Table 3-16 on page 3-132. The function pointed to by `func` is executed once when the signal is received. Handling is then returned to the default state.

The differences between the actions taken by the supplied standard interrupt dispatchers, `interrupt`, `interruptnsm`, `interruptf`, `interruptfnsm`, `interrupts`, `interruptsnsm`, `interruptcb`, and `interruptcbnsm`, are discussed under "signal.h" on page 3-18.

Error Conditions

The `signal` function returns `SIG_ERR` and sets `errno` to `SIG_ERR` if it does not recognize the requested signal.

Example

```
#include <signal.h>

signal (SIG_IRQ2, irq2_handler);    /* enable interrupt 2  */
signal (SIG_IRQ2, SIG_IGN);         /* disable interrupt 2 */
```

See Also

interrupt, raise

sin

sine

Synopsis

```
#include <math.h>
double sin (double x);
float sinf (float x);
long double sind (long double x);
```

Description

The sin functions return the sine of `x`. The input is interpreted as radians; the output is in the range [-1, 1].

Error Conditions

The input argument `x` must be in the domain [-1.647e6, 1.647e6] and the input argument for `sind` must be in the domain [-8.433e8, 8.433e8]. The functions return zero if `x` is outside their domain.

Example

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

y = sin (3.14159);      /* y = 0.0 */
x = sinf (3.14159);     /* x = 0.0 */
```

See Also

asin

sinh

hyperbolic sine

Synopsis

```
#include <math.h>

double sinh (double x);
float sinhf (float x);
long double sinhd (long double x);
```

Description

The sinh functions return the hyperbolic sine of `x`.

Error Conditions

The input argument `x` must be in the domain [-89.39, 89.39] for `sinhf`, and in the domain [-710.44, 710.44] for `sinhd`. If the input value is greater than the function's domain, then `HUGE_VAL` is returned, and if the input value is less than the domain, then `-HUGE_VAL` is returned.

Example

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

y = sinh (x);
z = sinhf (w);
```

See Also

cosh

sqrt

square root

Synopsis

```
#include <math.h>
double sqrt (double x);
float sqrtf (float x);
long double sqrtd (long double x);
```

Description

The sqrt functions return the positive square root of `x`.

Error Conditions

The sqrt functions return zero for negative input values and set `errno` to `EDOM` to indicate a domain error.

Example

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

y = sqrt (2.0);      /* y = 1.414..... */
x = sqrtf (2.0);     /* x = 1.414..... */
```

See Also

rsqrt

srand

random number seed

Synopsis

```
#include <stdlib.h>
void srand (unsigned int seed);
```

Description

The `srand` function is used to set the seed value for the `rand` function. A particular seed value always produces the same sequence of pseudo-random numbers.

Error Conditions

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

Example

```
#include <stdlib.h>

srand (22);
```

See Also

rand

strcat

concatenate strings

Synopsis

```
#include <string.h>
char *strcat (char *s1, const char *s2);
```

Description

The `strcat` function appends a copy of the null-terminated string pointed to by `s2` to the end of the null-terminated string pointed to by `s1`. It returns a pointer to the new `s1` string, which is null-terminated. The behavior of `strcat` is undefined if the two strings overlap.

Error Conditions

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

Example

```
#include <string.h>
char string1[50];

string1[0] = 'A';
string1[1] = 'B';
string1[2] = '\0';
strcat (string1, "CD");    /* new string is "ABCD" */
```

See Also

strncat

strchr

find first occurrence of character in string

Synopsis

```
#include <string.h>
char *strchr (const char *s1, int c);
```

Description

The `strchr` function returns a pointer to the first location in `s1`, a null-terminated string, that contains the character `c`.

Error Conditions

The `strchr` function returns a null pointer if `c` is not part of the string.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strchr (ptr1, 'E');
   /* ptr2 points to the E in TESTING */
```

See Also

memchr, strrchr

strcmp

compare strings

Synopsis

```
#include <string.h>
int strcmp (const char *s1, const char *s2);
```

Description

The `strcmp` function lexicographically compares the null-terminated strings pointed to by `s1` and `s2`. It returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

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

Example

```
#include <string.h>
char string1[50], string2[50];

if (strcmp (string1, string2))
    printf ("%s is different than %s \n", string1, string2);
```

See Also

memchr, strncmp

strcoll

compare strings

Synopsis

```
#include <string.h>
int strcoll (const char *s1, const char *s2);
```

Description

The `strcoll` function compares the string pointed to by `s1` with the string pointed to by `s2`. The comparison is based on the locale macro, `LC_COLLATE`. Because only the C locale is defined in the ADSP-21xxx run-time environment, the `strcoll` function is identical to the `strcmp` function. The function returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

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

Example

```
#include <string.h>
char string1[50], string2[50];

if (strcoll (string1, string2))
    printf ("%s is different than %s \n", string1, string2);
```

See Also

strcmp, strncmp

strcpy

copy from one string to another

Synopsis

```
#include <string.h>
char *strcpy (char *s1, const char *s2);
```

Description

The `strcpy` function copies the null-terminated string pointed to by `s2` into the space pointed to by `s1`. Memory allocated for `s1` must be large enough to hold `s2`, plus one space for the null character ('`\0`'). The behavior of `strcpy` is undefined if the two objects overlap or if `s1` is not large enough. The `strcpy` function returns the new `s1`.

Error Conditions

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

Example

```
#include <string.h>
char string1[50];

strcpy (string1, "SOMEFUN");
    /* SOMEFUN is copied into string1 */
```

See Also

memcpy, memmove, strncpy

strcspn

length of character segment in one string but not the other

Synopsis

```
#include <string.h>
size_t strcspn (const char *s1, const char *s2);
```

Description

The function `strcspn` returns the length of the initial segment of `s1` which consists entirely of characters not in the string pointed to by `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

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

Example

```
#include <string.h>
char *ptr1, *ptr2;
size_t len;

ptr1 = "Tried and Tested";
ptr2 = "aeiou";
len = strcspn (ptr1, ptr2);    /* len = 2 */
```

See Also

strlen, strspn

strerror

get string containing error message

Synopsis

```
#include <string.h>
char *strerror (int errnum);
```

Description

The function `strerror` returns a pointer to a string containing an error message by mapping the number in `errnum` to that string. Only one error is currently defined in the C environment for ADSP-21xxx processors.

Error Conditions

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

Example

```
#include <string.h>
char *ptr1;

ptr1 = strerror (1);
```

See Also

No references to this function.

strftime

format a broken-down time

Synopsis

```
#include <time.h>
size_t strftime(char *buf,
                size_t buf_size,
                const char *format,
                const struct tm *tm_ptr);
```

Description

The `strftime` function formats the broken-down time `tm_ptr` into the `char` array pointed to by `buf`, under the control of the format string `format`. At most, `buf_size` characters (including the null terminating character) are written to `buf`.

In a similar way as for `printf`, the format string consists of ordinary characters, which are copied unchanged to the `char` array `buf`, and zero or more conversion specifiers. A conversion specifier starts with the character `%` and is followed by a character that indicates the form of transformation required – the supported transformations are given below in Table 3-18. The `strftime` function only supports the "C" locale, and this is reflected in the table.

Table 3-18. Conversion Specifiers Supported by `strftime`

Conversion Specifier	Transformation	ISO/IEC 9899
`%a`	abbreviated weekday name	yes
`%A`	full weekday name	yes
`%b`	abbreviated month name	yes
`%B`	full month name	yes
`%c`	date and time presentation in the form of `DDD MMM dd hh:mm:ss yyyy`	yes

Table 3-18. Conversion Specifiers Supported by `strftime` (Cont'd)

Conversion Specifier	Transformation	ISO/IEC 9899
`%C`	century of the year	POSIX.2-1992 + ISO C99
`%d`	day of the month (`01 - 31`)	yes
`%D`	date represented as `mm/dd/yy`	POSIX.2-1992 + ISO C99
`%e`	day of the month, padded with a space character (cf `%d`)	POSIX.2-1992 + ISO C99
`%F`	date represented as `yyyy-mm-dd`	POSIX.2-1992 + ISO C99
`%h`	abbreviated name of the month (same as `%b`)	POSIX.2-1992 + ISO C99
`%H`	hour of the day as a 24-hour clock (`00-23`)	yes
`%I`	hour of the day as a 12-hour clock (`00-12`)	yes
`%j`	day of the year (`001-366`)	yes
`%k`	hour of the day as a 24-hour clock padded with a space (`0-23`)	no
`%l`	hour of the day as a 12-hour clock padded with a space (`0-12`)	no
`%m`	month of the year (`01-12`)	yes
`%m`	month of the year (`01-12`)	yes
`%M`	minute of the hour (`00-59`)	yes
`%n`	newline character	POSIX.2-1992 + ISO C99
`%p`	`AM` or `PM`	yes
`%P`	`am` or `pm`	no
`%r`	time presented as either `hh:mm:ss AM` or as `hh:mm:ss PM`	POSIX.2-1992 + ISO C99
`%R`	time presented as `hh:mm`	POSIX.2-1992 + ISO C99
`%S`	second of the minute (`00-61`)	yes

Table 3-18. Conversion Specifiers Supported by `strftime` (Cont'd)

Conversion Specifier	Transformation	ISO/IEC 9899
`%t`	tab character	POSIX.2-1992 + ISO C99
`%T`	time formatted as `%H:%M:%S`	POSIX.2-1992 + ISO C99
`%U`	week number of the year (week starts on Sunday) (`00-53`)	yes
`%w`	weekday as a decimal (`0-6`) (`0` if Sunday)	yes
`%W`	week number of the year (week starts on Sunday) (`00-53`)	yes
`%x`	date represented as `mm/dd/yy` (same as `%D`)	yes
`%X`	time represented as `hh:mm:ss`	yes
`%y`	year without the century (`00-99`)	yes
`%Y`	year with the century (`nnnn`)	yes
`%Z`	the time zone name, or nothing if the name cannot be determined	yes
`%%`	`%` character	yes

The current implementation of `time.h` does not support time zones and, therefore, the `%Z` specifier does not generate any characters.

The `strftime` function returns the number of characters (not including the terminating null character) that have been written to `buf`.

Error Conditions

The `strftime` function returns zero if more than `buf_size` characters are required to process the format string. In this case, the contents of the array `buf` will be indeterminate.

Example

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

extern void
print_time(time_t tod)
{
   char tod_string[100];

   strftime(tod_string,
            100,
            "It is %M min and %S secs after %l o'clock (%p)",
            gmtime(&tod));
   puts(tod_string);
}
```

See Also

ctime, gmtime, localtime, mktime

strlen

string length

Synopsis

```
#include <string.h>
size_t strlen (const char *s1);
```

Description

The `strlen` function returns the length of the null-terminated string pointed to by `s1` (not including the terminating null character).

Error Conditions

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

Example

```
#include <string.h>
size_t len;

len = strlen ("SOMEFUN");     /* len = 7 */
```

See Also

strcspn, strspn

strncat

concatenate characters from one string to another

Synopsis

```
#include <string.h>
char *strncat (char *s1, const char *s2, size_t n);
```

Description

The `strncat` function appends a copy of up to `n` characters in the null-terminated string pointed to by `s2` to the end of the null-terminated string pointed to by `s1`. It returns a pointer to the new `s1` string.

The behavior of `strncat` is undefined if the two strings overlap. The new `s1` string is terminated with a null character (`'\0'`).

Error Conditions

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

Example

```
#include <string.h>
char string1[50], *ptr;

string1[0] = '\0';
strncat (string1, "MOREFUN", 4);
    /* string1 equals "MORE" */
```

See Also

strcat

strncmp

compare characters in strings

Synopsis

```
#include <string.h>
int strncmp (const char *s1, const char *s2, size_t n);
```

Description

The `strncmp` function lexicographically compares up to `n` characters of the null-terminated strings pointed to by `s1` and `s2`. It returns a positive value if the `s1` string is greater than the `s2` string, a negative value if the `s2` string is greater than the `s1` string, and a zero if the strings are the same.

Error Conditions

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

Example

```
#include <string.h>
char *ptr1;

ptr1 = "TEST1";
if (strncmp (ptr1, "TEST", 4) == 0)
    printf ("%s starts with TEST\n", ptr1);
```

See Also

memcmp, strcmp

strncpy

copy characters from one string to another

Synopsis

```
#include <string.h>
char *strncpy (char *s1, const char *s2, size_t n);
```

Description

The `strncpy` function copies up to `n` characters of the null-terminated string pointed to by `s2` into the space pointed to by `s1`. If the last character copied from `s2` is not a null, the result does not end with a null. The behavior of `strncpy` is undefined if the two objects overlap. The `strncpy` function returns the new `s1`.

If the `s2` string contains fewer than `n` characters, the `s1` string is padded with the null character until all `n` characters have been written.

Error Conditions

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

Example

```
#include <string.h>
char string1[50];

strncpy (string1, "MOREFUN", 4);
                        /* MORE is copied into string1 */
string1[4] = '\0';      /* must null-terminate string1 */
```

See Also

memcpy, memmove, strcpy

strpbrk

find character match in two strings

Synopsis

```
#include <string.h>
char *strpbrk (const char *s1, const char *s2);
```

Description

The `strpbrk` function returns a pointer to the first character in `s1` that is also found in `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

In the event that no character in `s1` matches any in `s2`, a null pointer is returned.

Example

```
#include <string.h>
char *ptr1, *ptr2, *ptr3;

ptr1 = "TESTING";
ptr2 = "SHOP"
ptr3 = strpbrk (ptr1, ptr2);
   /* ptr3 points to the S in TESTING */
```

See Also

strspn

strrchr

find last occurrence of character in string

Synopsis

```
#include <string.h>
char *strrchr (const char *s1, int c);
```

Description

The `strrchr` function returns a pointer to the last occurrence of character `c` in the null-terminated input string `s1`.

Error Conditions

The `strrchr` function returns a null pointer if `c` is not found.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strrchr (ptr1, 'T');
   /* ptr2 points to the second T of TESTING */
```

See Also

memchr, strchr

strspn

length of segment of characters in both strings

Synopsis

```
#include <string.h>
size_t strspn (const char *s1, const char *s2);
```

Description

The `strspn` function returns the length of the initial segment of `s1` which consists entirely of characters in the string pointed to by `s2`. The string pointed to by `s2` is treated as a set of characters. The order of the characters in the string is not significant.

Error Conditions

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

Example

```
#include <string.h>
size_t len;
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = "ERST";
len = strspn (ptr1, ptr2);    /* len = 4 */
```

See Also

strcspn, strlen

strstr

find string within string

Synopsis

```
#include <string.h>
char *strstr (const char *s1, const char *s2);
```

Description

The `strstr` function returns a pointer to the first occurrence in the string pointed to by `s1` of the characters in the string pointed to by `s2`. This excludes the terminating null character in `s1`.

Error Conditions

If the string is not found, `strstr` returns a null pointer. If `s2` points to a string of zero length, `s1` is returned.

Example

```
#include <string.h>
char *ptr1, *ptr2;

ptr1 = "TESTING";
ptr2 = strstr (ptr1, "E");
   /* ptr2 points to the E in TESTING */
```

See Also

strchr

strtod

convert string to double

Synopsis

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr)
```

Description

The `strtod` function extracts a value from the string pointed to by `nptr`, and returns the value as a `double`. The `strtod` function expects `nptr` to point to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (–); and digits are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (–) followed by the hexadecimal prefix `0x` or `0X`. This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan. If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `strtod` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, a positive or negative (as appropriate) `HUGE_VAL` is returned. If the correct value results in an underflow, zero is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Example

```
#include <stdlib.h>
char *rem;
double dd;

dd = strtod ("2345.5E4 abc",&rem);
     /* dd = 2.3455E+7, rem = " abc" */

dd = strtod ("-0x1.800p+9,123",&rem);
     /* dd = -768.0, rem = ",123"   */
```

See Also

atof, strtol, strtoul

strtold

convert string to long double

Synopsis

```
#include <stdlib.h>
long double strtold(const char *nptr, char **endptr)
```

Description

The `strtold` function extracts a value from the string pointed to by `nptr`, and returns the value as a `long double`. The `strtold` function expects `nptr` to point to a string that represents either a decimal floating-point number or a hexadecimal floating-point number. Either form of number may be preceded by a sequence of whitespace characters (as determined by the `isspace` function) that the function ignores.

A decimal floating-point number has the form:

```
[sign] [digits] [.digits] [{e|E} [sign] [digits]]
```

The `sign` token is optional and is either plus (+) or minus (–); and digits are one or more decimal digits. The sequence of digits may contain a decimal point (.).

The decimal digits can be followed by an exponent, which consists of an introductory letter (`e` or `E`) and an optionally signed integer. If neither an exponent part nor a decimal point appears, a decimal point is assumed to follow the last digit in the string.

The form of a hexadecimal floating-point number is:

```
[sign] [{0x}|{0X}] [hexdigs] [.hexdigs] [{p|P} [sign] [digits]]
```

A hexadecimal floating-point number may start with an optional plus (+) or minus (–) followed by the hexadecimal prefix `0x` or `0X`. This character sequence must be followed by one or more hexadecimal characters that optionally contain a decimal point (.).

The hexadecimal digits are followed by a binary exponent that consists of the letter `p` or `P`, an optional sign, and a non-empty sequence of decimal digits. The exponent is interpreted as a power of two that is used to scale the fraction represented by the tokens `[hexdigs] [.hexdigs]`.

The first character that does not fit either form of number stops the scan. If `endptr` is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by `endptr`. If no conversion can be performed, the value of `nptr` is stored at the location pointed to by `endptr`.

Error Conditions

The `strtold` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`. If the correct value results in an overflow, a positive or negative (as appropriate) `LDBL_MAX` is returned. If the correct value results in an underflow, zero is returned. The `ERANGE` value is stored in `errno` in the case of either an overflow or underflow.

Example

```
#include <stdlib.h>
char *rem;
long double dd;

dd = strtold ("2345.5E4 abc",&rem);
     /* dd = 2.3455E+7, rem = " abc" */

dd = strtold ("-0x1.800p+9,123",&rem);
     /* dd = -768.0, rem = ",123"   */
```

See Also

atold, strtol, strtoul

strtok

convert string to tokens

Synopsis

```
#include <string.h>
char *strtok (char *s1, const char *s2);
```

Description

The `strtok` function returns successive tokens from the string `s1`, where each token is delimited by characters from `s2`.

A call to `strtok` with `s1` not NULL returns a pointer to the first token in `s1`, where a token is a consecutive sequence of characters not in `s2`. `s1` is modified in place to insert a null character at the end of the token returned. If `s1` consists entirely of characters from `s2`, NULL is returned.

Subsequent calls to `strtok` with `s1` equal to NULL return successive tokens from the same string. When the string contains no further tokens, NULL is returned. Each new call to `strtok` may use a new delimiter string, even if `s1` is NULL. If `s1` is NULL, the remainder of the string is converted into tokens using the new delimiter characters.

Error Conditions

The `strtok` function returns a null pointer if there are no tokens remaining in the string.

Example

```
#include <string.h>
static char str[] = "a phrase to be tested, today";
char *t;

t = strtok (str, " ");      /* t points to "a"            */
t = strtok (NULL, " ");     /* t points to "phrase"       */
```

```
t = strtok (NULL, ",");   /* t points to "to be tested" */
t = strtok (NULL, ".");   /* t points to " today"       */
t = strtok (NULL, ".");   /* t = NULL                   */
```

See Also

No references to this function.

strtol

convert string to long integer

Synopsis

```
#include <stdlib.h>
long int strtol (const char *nptr, char **endptr, int base);
```

Description

The `strtol` function returns as a `long int` the value represented by the string `nptr`. If `endptr` is not a null pointer, `strtol` stores a pointer to the unconverted remainder in `*endptr`.

The `strtol` function breaks down the input into three sections:

- White space (as determined by `isspace`)
- Initial characters
- Unrecognized characters including a terminating null character

The initial characters may be composed of an optional sign character, `0x` or `0X` if `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35, and their use is permitted only when those values are less than the value of `base`.

If `base` is zero, then the base is taken from the initial characters. A leading 0x indicates base 16; a leading 0 indicates base 8. For any other leading characters, base 10 is used. If `base` is between 2 and 36, it is used as a base for conversion.

Error Conditions

The `strtol` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer. If the correct value results in an overflow, positive or negative (as appropriate) `LONG_MAX` is returned. If the correct value results in an underflow, `LONG_MIN` is returned. `ERANGE` is stored in `errno` in the case of either overflow or underflow.

Example

```
#include <stdlib.h>
#define base 10
char *rem;
long int i;

i = strtol ("2345.5", &rem, base);
    /* i=2345, rem=".5" */
```

See Also

atoi, atol, strtod, strtoul

strtoul

convert string to unsigned long integer

Synopsis

```
#include <stdlib.h>
unsigned long int strtoul (const char *nptr, char **endptr, int base);
```

Description

The `strtoul` function returns as an `unsigned long int` the value represented by the string `nptr`. If `endptr` is not a null pointer, `strtoul` stores a pointer to the unconverted remainder in `*endptr`.

The `strtoul` function breaks down the input into three sections:

- White space (as determined by `isspace`)
- Initial characters
- Unrecognized characters including a terminating null character

The initial characters may comprise an optional sign character, `0x` or `0X`, when `base` is 16, and those letters and digits which represent an integer with a radix of `base`. The letters (`a-z` or `A-Z`) are assigned the values 10 to 35, and are permitted only when those values are less than the value of `base`.

If `base` is zero, then the base is taken from the initial characters. A leading 0x indicates base 16; a leading 0 indicates base 8. For any other leading characters, base 10 is used. If `base` is between 2 and 36, it is used as a base for conversion.

Error Conditions

The `strtoul` function returns a zero if no conversion can be made and a pointer to the invalid string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer. If the correct value results in an overflow, `ULONG_MAX` is returned. `ERANGE` is stored in `errno` in the case of overflow.

Example

```
#include <stdlib.h>
#define base 10

char *rem;
unsigned long int i;

i = strtoul ("2345.5", &rem, base);
    /* i = 2345, rem = ".5" */
```

See Also

atoi, atol, strtol

strxfrm

transform string using LC_COLLATE

Synopsis

```
#include <string.h>
size_t strxfrm (char *s1, const char *s2, size_t n);
```

Description

The strxfrm function transforms the string pointed to by s2 using the locale specific category LC_COLLATE. (See "setlocale" on page 3-185). It places the result in the array pointed to by s1.

The transformation is such that if s1 and s2 were transformed and used as arguments to strcmp, the result would be identical to the result derived from strcoll using s1 and s2 as arguments. However, since only C locale is implemented, this function does not perform any transformations other than the number of characters.

The string stored in the array pointed to by s1 is never more than n characters including the terminating NULL character. strxfrm returns 1. If this returned value is n or greater, the result stored in the array pointed to by s1 is indeterminate. s1 can be a NULL pointer if n is zero.

Error Conditions

The strxfrm function does not return an error condition.

Example

```
#include <string.h>
char string1[50];

strxfrm (string1, "SOMEFUN", 49);
    /* SOMEFUN is copied into string1 */
```

See Also

setlocale, strcmp, strcoll

system

send string to operating system

Synopsis

```
#include <stdlib.h>
int system (const char *string);
```

Description

The `system` function normally sends a string to the operating system. In the context of the ADSP-21xxx run-time environment, `system` always returns zero.

Error Conditions

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

Example

```
#include <stdlib.h>
system ("string");    /* always returns zero */
```

See Also

getenv

tan

tangent

Synopsis

```
#include <math.h>
double tan (double x);
float tanf (float x);
long double tand (long double x);
```

Description

The tan functions return the hyperbolic tangent of the argument `x`, where `x` is measured in radians.

Error Conditions

The domain of `tanf` is [-1.647e6, 1.647e6], and the domain for `tand` is [-4.21657e8 , 4.21657e8]. The functions return 0.0 if the input argument `x` is outside the respective domains.

Example

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

y = tan (3.14159/4.0);      /* y = 1.0 */
x = tanf (3.14159/4.0);     /* x = 1.0 */
```

See Also

atan, atan2

tanh

hyperbolic tangent

Synopsis

```
#include <math.h>
double tanh (double x);
float tanhf (float x);
long double tanhd (long double x);
```

Description

The tanh functions return the hyperbolic tangent of the argument `x`, where `x` is measured in radians.

Error Conditions

The tanh functions do not return an error condition.

Example

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

y = tanh (x);
z = tanhf (w);
```

See Also

cosh, sinh

time

calendar time

Synopsis

```
#include <time.h>
time_t time(time_t *t);
```

Description

The `time` function returns the current calendar time which measures the number of seconds that have elapsed since the start of a known epoch. As the calendar time cannot be determined in this implementation of `time.h`, a result of `(time_t) -1` is returned. The function's result is also assigned to its argument, if the pointer to `t` is not a null pointer.

Error Conditions

The `time` function will return the value `(time_t) -1` if the calendar time is not available.

Example

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

if (time(NULL) == (time_t) -1)
    printf("Calendar time is not available\n");
```

See Also

ctime, gmtime, localtime

tolower

convert from uppercase to lowercase

Synopsis

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

Description

The `tolower` function converts the input character to lowercase if it is uppercase; otherwise, it returns the character.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    if (isupper (ch))
        printf ("tolower=%#04x", tolower (ch));
    putchar ('\n');
}
```

See Also

islower, isupper, toupper

toupper

convert from lowercase to uppercase

Synopsis

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

Description

The `toupper` function converts the input character to uppercase if it is in lowercase; otherwise, it returns the character.

Error Conditions

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

Example

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

for (ch = 0; ch <= 0x7f; ch++) {
    printf ("%#04x", ch);
    if (islower (ch))
        printf ("toupper=%#04x", toupper (ch));
    putchar ('\n');
}
```

See Also

islower, isupper, tolower

va_arg

get next argument in variable-length list of arguments

Synopsis

```
#include <stdarg.h>
void va_arg (va_list ap, type);
```

Description

The `va_arg` macro is used to walk through the variable length list of arguments to a function.

After starting to process a variable-length list of arguments with `va_start`, call `va_arg` with the same `va_list` variable to extract arguments from the list. Each call to `va_arg` returns a new argument from the list.

Substitute a `type` name corresponding to the type of the next argument for the `type` parameter in each call to `va_arg`. After processing the list, call `va_end`.

The header file `stdarg.h` defines a pointer type called `va_list` that is used to access the list of variable arguments.

The function calling `va_arg` is responsible for determining the number and types of arguments in the list. It needs this information to determine how many times to call `va_arg` and what to pass for the type parameter each time. There are several common ways for a function to determine this type of information. The standard C `printf` function reads its first argument looking for %-sequences to determine the number and types of its extra arguments. In the example below, all of the arguments are of the same type (`char*`), and a termination value (NULL) is used to indicate the end of the argument list. Other methods are also possible.

If a call to `va_arg` is made after all arguments have been processed, or if `va_arg` is called with a type parameter that is different from the type of the next argument in the list, the behavior of `va_arg` is undefined.

Error Conditions

The `va_arg` macro does not return an error condition.

Example

```
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>

char *concat(char *s1, ...)
{
    int len = 0;
    char *result;
    char *s;
    va_list ap;

    va_start (ap, s1);
    s = s1;
    while (s) {
        len += strlen (s);
        s = va_arg (ap, char *);
    }
    va_end (ap);

    result = malloc (len + 7);
    if  (!result)
        return result;
    *result = ' ';
    va_start (ap, s1);
    s = s1;
    while (s) {
        strcat (result, s);
        s = va_arg (ap, char *);
    }
    va_end (ap);
    return result;
}
```

See Also

va_end, va_start

va_end

finish variable-length argument list processing

Synopsis

```
#include <stdarg.h>
void va_end (va_list ap);
```

Description

The `va_end` macro can only be invoked after the `va_start` macro has been invoked. A call to `va_end` concludes the processing of a variable-length list of arguments that was begun by `va_start`.

Error Conditions

The `va_end` macro does not return an error condition.

Example

See "va_arg" on page 3-233

See Also

va_arg, va_start

va_start

initialize the variable-length argument list processing

Synopsis

```
#include <stdarg.h>
void va_start (va_list ap, parmN);
```

Description

The `va_start` macro is used to start processing variable arguments in a function declared to take a variable number of arguments. The first argument to `va_start` should be a variable of type `va_list`, which is used by `va_arg` to walk through the arguments.

The second argument is the name of the last *named* parameter in the function's parameter list; the list of variable arguments immediately follows this parameter. The `va_start` macro must be invoked before either the `va_arg` or `va_end` macro can be invoked.

Error Conditions

The `va_start` macro does not return an error condition.

Example

See "va_arg" on page 3-233

See Also

va_arg, va_end

4 DSP LIBRARY FOR ADSP-2106X AND ADSP-21020 PROCESSORS

This chapter describes the DSP run-time library for ADSP-2106x and ADSP-21020 processors which contains a collection of functions that provide services commonly required by signal processing applications; these functions are in addition to the C/C++ run-time library functions that are described in Chapter 3, "C/C++ Run-Time Library". The services provided by the DSP library functions include support for interrupt handling, signal processing, and access to hardware registers. All these services are Analog Devices extensions to ANSI standard C.

The chapter contains:

- "DSP Run-Time Library Guide" (starting on on page 4-2) contains introductory information about the Analog Devices' special header files and built-in functions that are included with this release of the `cc21k` compiler.
- "DSP Run-Time Library Reference" (starting on on page 4-15) contains the complete reference information for each DSP run-time library function included with this release of the `cc21k` compiler.

For more information on the algorithms on which many of the DSP run-time 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.

DSP Run-Time Library Guide

The DSP run-time library contains routines that you can call from your source program. This section describes how to use the library and provides information on the following topics:

- "Calling DSP Library Functions" on page 4-2
- "Linking DSP Library Functions" on page 4-3
- "Working With Library Source Code" on page 4-3
- "DSP Header Files" on page 4-4
- "Built-In DSP Functions" on page 4-13

For information on the contents of the DSP library, see "DSP Run-Time Library Reference" on page 4-15 and on-line Help.

Calling DSP Library Functions

To use a DSP library function, call the function by name and give the appropriate arguments. The names and arguments for each function are described in the function's reference page in the section "DSP Run-Time Library Reference" on page 4-15.

Like other functions you use, library functions should be declared. Declarations are supplied in header files, as described in the section, "Working With Library Source Code" on page 4-3.

C++ namespace prefixing is not supported when calling a DSP library function. All DSP library functions are in the C++ global namespace.

The function names are C function names. If you call C run-time library functions from an assembly language program, you must use the assembly version of the function name, which is the func-

tion name prefixed with an underscore. For more information on naming conventions, see "C/C++ and Assembly Interface" on page 1-209.

You can use the archiver, described in the *VisualDSP++ 4.0 Linker and Utilities Manual*, to build library archive files of your own functions.

Linking DSP Library Functions

When your C code calls a DSP run-time library function, the call creates a reference that the linker resolves when linking your program. One way to direct the linker to the location of the DSP library is to use the default Linker Description File (`ADSP-21<your_target>.ldf`). The default Linker Description File automatically direct the linker to the library `libdsp.dlb` in the `21k\lib` subdirectory of your VisualDSP++ installation. If not using the default `.LDF` file, then either add `libdsp.dlb` to the `.LDF` file used for your project, or alternatively use the compiler's `-ldsp` switch to specify that `libdsp.dlb` is to be added to the link line.

When compiling for the ADSP-21020 processor, the name of the DSP run-time library is `libdsp020.dlb`.

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 `...\21k\lib\src`. The directory contains the source for the C run-time library, for the DSP run-time library, and for the I/O run-time library, as well as the source for the main program start-up functions. If you do not intend to modify any of the run-time library functions, you can delete this directory and its contents to conserve disk space.

The source code is provided so you can customize specific functions for your own needs. To modify these files, you need proficiency in ADSP-21xxx assembly language and an understanding of the run-time environment, as explained in "C/C++ Run-Time Model and Environment" on page 1-175). 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.

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 uses the prototypes to check that each function is called with the correct arguments. The following sections briefly describe the DSP header files supplied with this release of the `cc21k` compiler.

21020.h — ADSP-21020 DSP Functions

The `21020.h` header file includes the ADSP-21020 processor-specific functions of the DSP library, such as `poll_flag_in()`, `timer_set()`, and `idle()`. The `timer_set()`, `timer_on()`, and `timer_off()` functions are also available as inline functions.

21060.h — ADSP-2106x DSP Functions

The `21060.h` header file includes the ADSP-2106x processor-specific functions of the DSP library, such as `poll_flag_in()`, `timer_set()`, and `idle()`. The `timer_set()`, `timer_on()`, and `timer_off()` functions are also available as inline functions.

21065L.h — ADSP-21065L DSP Functions

The `21065L.h` header file includes the ADSP-21065L processor-specific functions of the DSP library, such as `poll_flag_in()` and `idle()`. The header file also includes support for the two programmable timers in the form of inline functions.

asm_sprt.h — Mixed C/Assembly Support

The `asm_sprt.h` header file consists of the ADSP-21xxx assembly language macros, not C functions. They are used in your assembly routines that interface with C functions. For more information, see "Using Mixed C/C++ and Assembly Support Macros" on page 1-214.

cmatrix.h — Complex Matrix Functions

The `cmatrix.h` header file contains prototypes for functions that perform basic arithmetic between two complex matrices, and also between a complex matrix and a complex scalar. The supported complex types are described under the header file `complex.h`.

comm.h — A-law and μ-law Companders

The `comm.h` header file includes the voice-band compression and expansion communication functions of the DSP library.

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 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;
```

Additional support for complex numbers is available via the `cmatrix.h` and `cvector.h` header files.

cvector.h — Complex Vector Functions

The `cvector.h` header file contains functions for basic arithmetical operations on vectors of type `complex_float`, `complex_double`, and `complex_long_double`. Support is provided for the dot product operation, as well as for adding, subtracting, and multiplying a vector by either a scalar or vector.

Header Files Defining Processor-Specific System Register Bits

The following header files define symbolic names for processor-specific system register bits. They also contain symbolic definitions for the IOP register address memory and IOP control/status register bits.

The processor-specific header files are:

def21020.h	ADSP-21020 Bit Definitions
def21060.h	ADSP-21060 Bit Definitions
def21061.h	ADSP-21061 Bit Definitions
def21062.h	ADSP-21062 Bit Definitions
def21065L.h	ADSP-21065L Bit Definitions

Header Files To Allow Access to Memory Mapped Registers From C/C++ Code

In order to allow safe access to memory-mapped registers from C/C++ code, the header files listed below are supplied. Each memory-mapped register's name is prefixed with "p" and is cast appropriately to ensure the code is generated correctly. For example, SYSCON is defined as follows:

```
#define pSYSCON ((volatile unsigned int *) 0x00)
```

and can be used as:

```
*pSYSCON |= 0x6000;
```

This method of accessing memory-mapped registers should be used in preference to using asm statements.

Supplied header files are:

Cdef21060.h	Cdef21061.h	Cdef21062.h	Cdef21065l.h

dma.h — DMA Support Functions

The dma.h header file provides definitions and setup, status, enable and disable functions for DMA operations.

filters.h — DSP Filters

The `filters.h` header file includes the signal processing filter functions of the DSP library.

macros.h — Circular Buffers

The `macro.h` header file consists of ADSP-21xxx assembly language macros, not C functions. Some are used to manipulate the circular buffer features of the ADSP-21xxx processors.

math.h — Math Functions

The standard math functions defined in the `math.h` header file have been augmented by implementations for the `float` and `long double` data types and some additional functions that are Analog Devices extensions to the ANSI standard.

Table 4-1 provides a summary of the additional library functions defined by the `math.h` header file.

Table 4-1. Math Library - Additional Functions

Description	Prototype

Anti-log	`double alog (double x);`
	`float alogf (float x);`
	`long double alogd (long double x);`

Base 10 Anti-log	`double alog10 (double x);`
	`float alog10f (float x);`
	`long double alog10d (long double x);`

sign copy	`double copysign (double x, double y);`
	`float copysignf (float x, float y);`
	`long double copysignd (long double x, long double y);`

cotangent	`double cot (double x);`
	`float cotf (float x);`
	`long double cotd (long double x);`

Table 4-1. Math Library - Additional Functions (Cont'd)

Description	Prototype

average	`double favg (double x, double y);`
	`float favgf (float x, float y);`
	`long double favgd (long double x, long double y);`

clip	`double fclip (double x, double y);`
	`float fclipf (float x, float y);`
	`long double fclipd (long double x, long double y);`

detect Infinity	`int isinf (double x);`
	`int isinff (float x);`
	`int isinfd (long double x);`

detect NaN	`int isnan (double x);`
	`int isnanf (float x);`
	`int isnand (long double x);`

maximum	`double fmax (double x, double y);`
	`float fmaxf (float x, float y);`
	`long double fmaxd (long double x, long double y);`

minimum	`double fmin (double x, double y);`
	`float fminf (float x, float y);`
	`long double fmind (long double x, long double y);`

reciprocal of square root	`double rsqrt (double x, double y);`
	`float rsqrtf (float x, float y);`
	`long double rsqrtd (long double x, long double y);`

matrix.h — Matrix Functions

The `matrix.h` header file declares a number of function prototypes associated with basic arithmetic operations on matrices of type `float`, `double`, and `long double`. The header file contains support for arithmetic between two matrices, and between a matrix and a scalar.

saturate.h — Saturation Mode Arithmetic

The `saturate.h` header file defines the interface for the saturated arithmetic operations. See "Saturated Arithmetic" on page 1-149 for further information.

sport.h — Serial Port Support Functions

The `sport.h` header file provides definitions and setup, enable, and disable functions for the ADSP-2106x DSP serial ports.

stats.h — Statistical Functions

The `stats.h` header file includes various statistics functions of the DSP library, such as `mean()` and `autocorr()`.

sysreg.h — Register Access

The `sysreg.h` header file defines a set of built-in functions that provide efficient access to the SHARC system registers from C. The supported functions are fully described in the section "Access to System Registers" on page 1-103.

trans.h — Fast Fourier Transforms

The `trans.h` header file includes the Fast Fourier Transform (FFT) functions of the DSP library. Note that `cfftN` stands for the entire family of Fast Fourier Transform functions `cfft65536`, `cfft32768`, etc.

The `cfftN` functions compute the fast Fourier transform (FFT) of their N-point complex input signal. The `ifftN` functions compute the inverse fast Fourier transform of their N-point complex input signal. The input to each of these functions is two `float` arrays (real and imaginary) of N elements. The routines output two N-element arrays.

If you only wish to input the real part of a signal, ensure that the imaginary input array is filled with zeros before calling the function.

The functions first bit-reverse the input arrays and then process them with an optimized block floating-point FFT (or IFFT) routine.

The `rfftN` functions work like `cfftN` functions, except they operate on input arrays of real data only. This is equivalent to a `cfftN` whose imaginary input component is set to zero.

vector.h — Vector Functions

The `vector.h` header file contains functions for operating on vectors of type `float`, `double` and `long double`. Support is provided for the dot product operation and well as for adding, subtracting, and multiplying a vector by either a scalar or vector. Similar support for the complex data types is defined in the header file `cvector.h`.

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 4-2.

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 4-2. Window Generator Functions

Description	Prototype
generate bartlett window	`void gen_bartlett (float w[], int a, int n)`
generate blackman window	`void gen_blackman (float w[], int a, int n)`
generate gaussian window	`void gen_gaussian (float w[], float alpha, int a, int n)`
generate hamming window	`void gen_hamming (float w[], int a, int n)`
generate hanning window	`void gen_hanning (float w[], int a, int n)`
generate harris window	`void gen_harris (float w[], int a, int n)`
generate kaiser window	`void gen_kaiser (float w[], float beta, int a, int n)`
generate rectangular window	`void gen_rectangular (float w[], int a, int n)`
generate triangle window	`void gen_triangle (float w[], int a, int n)`
generate von hann window	`void gen_vonhann (float w[], int a, int n)`

Built-In DSP Functions

The C/C++ compiler supports built-in functions (also known as intrinsic functions) that enable efficient use of hardware resources. Knowledge of these functions is built into the compiler. Your program uses them via normal function call syntax. The compiler notices the invocation and replaces a call to a DSP library function with one or more machine instructions, just as it does for normal operators like "+" and "*".

Built-in functions are declared in system header files and have names which begin with double underscores, `__builtin`.

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

These functions are specific to individual architectures. The built-in DSP library functions supported at this time on the ADSP-2106x and ADSP-20120 architectures are listed in Table 4-3. Refer to "Using Compiler Built-In C Library Functions" on page 3-30 for more information on this topic.

Table 4-3. Built-in DSP Functions

`avg`	`clip`	`copysign`	`copysignf`
`favg`	`favgf`	`fmax`	`fmaxf`
`fmin`	`fminf`	`labs`	`lavg`
`lclip`	`lmax`	`lmin`	`max`
`min`			

Functions `copysign`, `favg`, `fmax`, and `fmin` are compiled as a built-in function only if `double` is the same size as `float`.

Use the `-no-builtin` compiler switch (see on page 1-37) to disable this feature.

The compiler also supports a set of built-in functions for which no inline machine instructions are substituted. This set of built-in functions is characterized by defining one or more pointers in their argument list.

For this set of built-in functions, the compiler relaxes the normal rule whereby any pointer that is passed to a library function must address Data Memory (DM). The compiler recognizes when certain pointers address Program Memory (PM) and generates a call to an appropriate version of the run-time library function. Table 4-4 lists library functions that may be called with pointers that address Program Memory.

Table 4-4. Library Functions Called with Pointers

histogram	matmaddf	matmmltf
matmsubf	matsaddf	matsmltf
matssubf	meanf	rmsf
transpmf	varf	zero_crossf

Use the `-no-builtin` compiler switch (see on page 1-37) to disable this feature.

DSP Run-Time Library Reference

The DSP 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. Note the following items that apply to all the functions in the library.

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

Function Benchmarks and Specifications. All functions have been timed from setup, to invocation, to results storage of returned value. This includes all register storing, parameter passing, and so on. Most functions execute slightly faster if you pass constants as arguments instead of variables.

Restrictions. When polymorphic functions are used and the function returns a pointer to program memory, cast the output of the function to pm. For example, `(char pm *)`

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

Name and Purpose of the function

Synopsis – Required header file and functional prototype

Description – Function specification

Error Conditions – Method that the functions use to indicate an error

Example –Typical function usage

See Also – Related functions

a_compress

A-law compression

Synopsis

```
#include <comm.h>
int a_compress (int x);
```

Description

The `a_compress` function takes a linear 13-bit signed speech sample and compresses it according to ITU recommendation G.711. The value returned is an 8-bit sample that can be sent directly to an A-law codec.

Error Conditions

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

Example

```
#include <comm.h>
int sample, compress;

compress = a_compress (sample);
```

See Also

a_expand, mu_compress

a_expand

A-law expansion

Synopsis

```
#include <comm.h>
int a_expand (int compress_x);
```

Description

The `a_expand` function takes an 8-bit compressed speech sample and expands it according to ITU recommendation G.711 (A-law definition). The value returned is a linear 13-bit signed sample.

Error Conditions

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

Example

```
#include <comm.h>
int compressed_sample, expanded;

expanded = a_expand (compressed_sample);
```

See Also

a_compress, mu_expand

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.

Error Conditions

The input argument `x` for `alogf` must be in the domain [-87.3, 88.7] and the input argument for `alogd` must be in the domain [-708.2, 709.1]. The functions return `HUGE_VAL` if `x` is greater than the domain, and they return 0.0 if `x` is less than the domain.

Example

```
#include <math.h>

double x = 1.0;
double y;
y = alog(x);            /* y = 2.71828... */
```

See Also

alog10, exp, log, pow

alog10

base 10 anti-log

Synopsis

```
#include <math.h>

float alog10f (float x);
double alog10 (double x);
long double alog10d (long double x);
```

Description

The 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))`.

Error Conditions

The input argument `x` for `alog10f` must be in the domain [-37.9 , 38.5] and the input argument for `alog10d` must be in the domain [-307.57, 308.23]. The functions return `HUGE_VAL` if `x` is greater than the domain, and they return 0.0 if `x` is less than the domain.

Example

```
#include <math.h>

double x = 1.0;
double y;
y = alog10(x);          /* 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. The phase of a Cartesian number is computed as:

$$c = atan\left(\frac{\mathrm{Im}(a)}{\mathrm{Re}(a)}\right)$$

Error Conditions

The arg functions return a zero if `a.re` <> 0 and `a.im` = 0.

Example

```
#include <complex.h>

complex_float x = {0.0,1.0};
float r;
r = argf(x);       /* r = π/2 */
```

See Also

atan2, cartesian, polar

autocoh

autocoherence

Synopsis

```
#include <stats.h>

float *autocohf (float dm out[],
                 const float dm in[], int samples, int lags);

double *autocoh (double dm out[],
                 const double dm in[], int samples, int lags);

long double *autocohd (long double dm out[],
                       const long double dm in[],
                       int samples, int lags);
```

Description

The autocoh functions compute the autocoherence of the floating-point input, `in[]`, which contain `samples` values. The autocoherence of an input signal is its autocorrelation minus its mean squared. The functions return a pointer to the output array `out[]` of length `lags`.

Error Conditions

The autocoh functions do not return an error condition.

Example

```
#include <stats.h>
#define SAMPLES 1024
#define LAGS      16

double excitation[SAMPLES];
double response[LAGS];
int lags = LAGS;

autocoh (response, excitation, SAMPLES, lags);
```

See Also

autocorr, crosscoh, crosscorr

autocorr

autocorrelation

Synopsis

```
#include <stats.h>

float *autocorrf (float dm out[], const float dm in[],
                  int samples, int lags);

double *autocorr (double dm out[], const double dm in[],
                  int samples, int lags);

long double *autocorrd (long double dm out[],
                        const long double dm in[],
                        int samples, int lags);
```

Description

The autocorr functions perform an autocorrelation of a signal. Autocorrelation is the cross-correlation of a signal with a copy of itself. It provides information about the time variation of the signal. The signal to be autocorrelated is given by the `in[]` input array. The number of samples of the autocorrelation sequence to be produced is given by `lags`. The length of the input sequence is given by `samples`. The functions return a pointer to the `out[]` output data array of length `lags`.

Autocorrelation is used in digital signal processing applications such as speech analysis.

Error Conditions

The autocorr functions do not return an error condition.

Example

```
#include <stats.h>
#define SAMPLES 1024
```

```
#define LAGS      16

double excitation[SAMPLES];
double response[LAGS];
int lags = LAGS;

autocorr (response, excitation, SAMPLES, lags);
```

See Also

autocoh, crosscoh, crosscorr

biquad

biquad filter section

Synopsis

```
#include <filters.h>
float biquad (float sample,
              const float pm coeffs[],
              float dm state[],
              int sections);
```

Description

The `biquad` function implements a cascaded biquad filter. The function produces the filtered response of its input data. The parameter `sections` specifies the number of biquad sections.

Each biquad section is represented by five coefficients that must be stored in the order `B2`, `B1`, `B0`, `A2`, `A1`. The value of `A0` is assumed to be `1.0`, and `A1` and `A2` should be scaled accordingly. The coefficients should be stored in the array `coeffs` which must be located in program memory (PM). The definition of the `coeffs` array is:

```
float pm coeffs[5*sections];
```

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]`.

The `state` array holds two delay elements per section. It also has one extra location that holds an internal pointer. The total length must be equal to `2*sections + 1`. The definition is:

```
float dm state[2*sections + 1];
```

Each filter has its own state array which should be initialized to zero before calling the `biquad` function for the first time, and should not otherwise be modified by the calling program.

Error Conditions

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

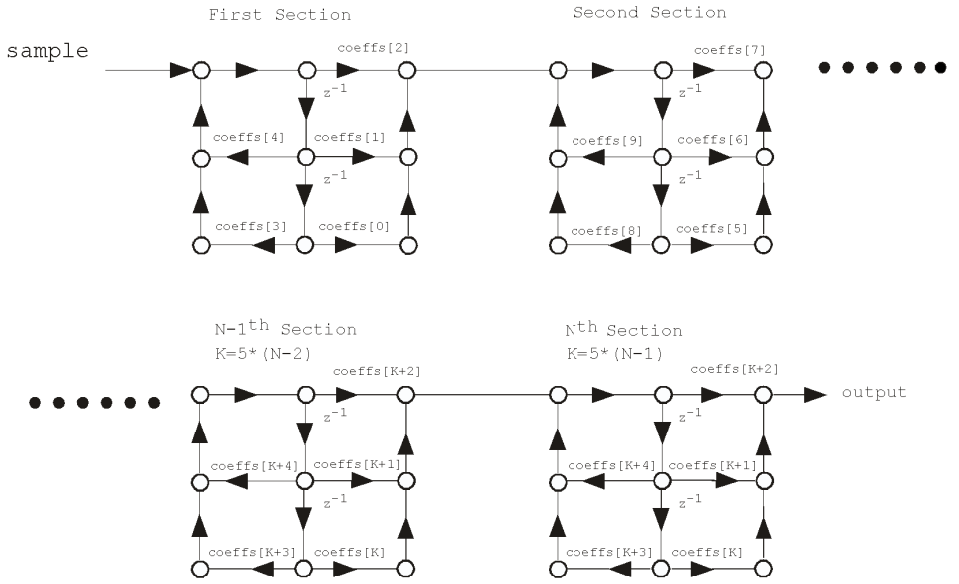
Example

```
#include <filters.h>
#define TAPS 9

float sample, output, state[2*TAPS+1];
float pm coeffs[5*TAPS];
int i;

for (i = 0; i < 2*TAPS+1; i++)
    state[i] = 0;/* initialize state array */

output = biquad (sample, coeffs, state, TAPS);
```



N = the number of biquad sections.

The algorithm shown here is adapted from Oppenheim, Alan V. and Ronald Schafer, *Digital Signal Processing*, Englewood Cliffs, New Jersey: Prentice Hall, 1975.

See Also

fir, iir

cabs

complex absolute value

Synopsis

```
#include <complex.h>
float cabsf (complex_float z);
double cabs (complex_double z);
long double cabsd (complex_long_double z);
```

Description

The cabs functions return the floating-point absolute value of their complex input.

The absolute value of a complex number is evaluated with the following formula.

$$y = \sqrt{((\mathrm{Re}(z))^2 + (\mathrm{Im}(z))^2)}$$

Error Conditions

The cabs functions do not return an error condition.

Example

```
#include <complex.h>
complex_float cnum;
float answer;

cnum.re = 12.0;
cnum.im = 5.0;

answer = cabsf (cnum);      /* answer = 13.0 */
```

See Also

fabs, labs

cadd

complex addition

Synopsis

```
#include <complex.h>
complex_float caddf (complex_float a, complex_float b);
complex_double cadd (complex_double a, complex_double b);
complex_long_double caddd (complex_long_double a,
                           complex_long_double b);
```

Description

The cadd functions add the two complex values `a` and `b` together, and return the result.

Error Conditions

The cadd functions do not return any error conditions.

Example

```
#include <complex.h>

complex_double x = {9.0,16.0};
complex_double y = {1.0,-1.0};
complex_double z;

z = cadd (x,y);     /* z.re = 10.0, z.im = 15.0 */
```

See Also

cdiv, cmlt, csub

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 it returns as the function's result, and a phase that is returned via the second argument `phase`.

The formula for converting from Cartesian to polar notation is given by:

```
magnitude = cabs(a)
phase = arg(a)
```

Error Conditions

The cartesian functions return a zero for the phase if `a.re` <> 0 and `a.im` = 0.

Example

```
#include <complex.h>

complex_float point = {-2.0 , 0.0};
float phase;
```

```
float mag;
mag = cartesianf (point,&phase);      /* mag = 2.0, phase = π */
```

See Also

arg, cabs, polar

cdiv

complex division

Synopsis

```
#include <complex.h>

complex_float cdivf (complex_float a, complex_float b);
complex_double cdiv (complex_double a, complex_double b);
complex_long_double cdivd (complex_long_double a,
                           complex_long_double b);
```

Description

The cdiv functions compute the complex division of complex input `a` by complex input `b`, and return the result.

Error Conditions

The cdiv functions set both the real and imaginary parts of the result to Infinity if `b` is equal to `(0.0,0.0)` .

Example

```
#include <complex.h>

complex_double x = {3.0,11.0};
complex_double y = {1.0, 2.0};
complex_double z;

z = cdiv (x,y);     /* z.re = 5.0, z.im = 1.0 */
```

See Also

cadd, cmlt, csub

cexp

complex exponential

Synopsis

```
#include <complex.h>
complex_float cexpf (complex_float z);
complex_double cexp (complex_double z);
complex_long_double cexpd (complex_long_double z);
```

Description

The cexp functions compute the complex exponential value e to the power of the first argument. The exponential of a complex value is evaluated with the following formula.

```
Re(y) = exp (Re(z)) * cos (Im(z));
Im(y) = exp (Re(z)) * sin (Im(z));
```

Error Conditions

For underflow errors, the cexp functions return zero.

Example

```
#include <complex.h>

complex_float cnum;
complex_float answer;

cnum.re = 1.0;
cnum.im = 0.0;

answer = cexpf (cnum);      /* answer = (2.7182 + 0i) */
```

See Also

log, pow

cfftN

N-point complex input FFT

Synopsis

```
#include <trans.h>

float *cfft65536 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft32768 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft16384 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft8192 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft4096 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft2048 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft1024 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *cfft512  (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);
```

```
float *cfft256  (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *cfft128  (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *cfft64   (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *cfft32   (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *cfft16   (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);

float *cfft8    (const float dm real_input[],
                 const float dm imag_input[],
                 float dm real_output[], float dm imag_output[]);
```

Description

Each of these 14 `cfftN` functions computes the N-point radix-2 fast Fourier transform (CFFT) of its floating-point input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are fourteen distinct functions in this set. They all perform the same function with the same type and number of arguments. The only difference between them is the size of the arrays they operate on. Call a particular function by substituting the number of points for N, as in

```
cfft8 (r_inp, i_inp, r_outp, i_outp);
```

The input to `cfftN` are two floating-point arrays of N points. The array `real_input` contains the real components of the complex signal, and the array `imag_input` contains the imaginary components.

If there are fewer than N actual data points, you must pad the arrays with zeros to make N samples. However, better results occur with less zero padding. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data.

If the input data can be overwritten, then the `cfftN` functions allow the array `real_input` to share the same memory as the array `real_output`, and `imag_input` to share the same memory as `imag_output`. This would improve memory usage, but at the cost of some run-time performance.

The `cfftN` functions return a pointer to the `real_output` array.

Error Conditions

The `cfftN` functions do not return any error conditions.

Example

```
#include <trans.h>
#define N 2048

float real_input[N], imag_input[N];
float real_output[N], imag_output[N];

   /* Real input array is filled from a converter or other source */

cfft2048 (real_input, imag_input, real_output, imag_output);
   /* Arrays are filled with FFT data */
```

See Also

ifftN, rfftN

cmatmadd

complex matrix + matrix addition

Synopsis

```
#include <cmatrix.h>

complex_float *cmatmaddf (complex_float dm *output,
                          const complex_float dm *a,
                          const complex_float dm *b,
                          int rows, int cols);

complex_double *cmatmadd (complex_double dm *output,
                          const complex_double dm *a,
                          const complex_double dm *b,
                          int rows, int cols);

complex_long_double *cmatmaddd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          const complex_long_double dm *b,
                          int rows, int cols);
```

Description

The cmatmadd functions perform a complex matrix addition of the input matrix `a[][]` with input complex matrix `b[][]`, and store the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The cmatmadd functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (double_complex *) (&a);
complex_double b[ROWS][COLS], *b_p = (double_complex *) (&b);
complex_double c[ROWS][COLS], *res_p = (double_complex *) (&c);

cmatmadd (res_p, a_p, b_p, ROWS, COLS);
```

See Also

cmatmsub, cmatsadd

cmatmmlt

complex matrix * matrix multiplication

Synopsis

```
#include <cmatrix.h>

complex_float *cmatmmltf (complex_float dm *output,
                          const complex_float dm *a,
                          const complex_float dm *b,
                          int a_rows, int a_cols, int b_cols);

complex_double *cmatmmlt (complex_double dm *output,
                          const complex_double dm *a,
                          const complex_double dm *b,
                          int a_rows, int a_cols, int b_cols);

complex_long_double *cmatmmltd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          const complex_long_double dm *b,
                          int a_rows, int a_cols, int b_cols);
```

Description

The cmatmmlt functions perform a complex matrix multiplication of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[a_rows][a_cols]`, `b[a_cols][b_cols]`, and `output[a_rows][b_cols]`. The functions return a pointer to the output matrix.

Complex matrix multiplication is defined by the following algorithm:

$$\mathrm{Re}(c_{i,j}) = \sum_{l=0}^{a_cols-1} (\mathrm{Re}(a_{i,l}) * \mathrm{Re}(b_{l,j}) - \mathrm{Im}(a_{i,l}) * \mathrm{Im}(b_{l,j}))$$

$$\mathrm{Im}(c_{i,j}) = \sum_{l=0}^{a_cols-1} (\mathrm{Re}(a_{i,l}) * \mathrm{Im}(b_{l,j}) + \mathrm{Im}(a_{i,l}) * \mathrm{Re}(b_{l,j}))$$

where i={0,1,2,...,a_rows-1}, j={0,1,2,...,b_cols-1}

Error Conditions

The cmatmmlt functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS_1 4
#define COLS_1 8
#define COLS_2 2

complex_double a[ROWS_1][COLS_1], *a_p = (double_complex *) (&a);
complex_double b[COLS_1][COLS_2], *b_p = (double_complex *) (&b);
complex_double c[ROWS_1][COLS_2], *r_p = (double_complex *) (&c);

cmatmadd (r_p, a_p, b_p, ROWS_1, COLS_1, COLS_2);
```

See Also

cmatsmlt

cmatmsub

complex matrix - matrix subtraction

Synopsis

```
#include <cmatrix.h>

complex_float *cmatmsubf (complex_float dm *output,
                          const complex_float dm *a,
                          const complex_float dm *b,
                          int rows, int cols);

complex_double *cmatmsub (complex_double dm *output,
                          const complex_double dm *a,
                          const complex_double dm *b,
                          int rows, int cols);

complex_long_double *cmatmsubd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          const complex_long_double dm *b,
                          int rows, int cols);
```

Description

The cmatmsub functions perform a complex matrix subtraction between the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The cmatmsub functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
```

```
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (double_complex *) (&a);
complex_double b[ROWS][COLS], *b_p = (double_complex *) (&b);
complex_double c[ROWS][COLS], *res_p = (double_complex *) (&c);

cmatmsub (res_p, a_p, b_p, ROWS, COLS);
```

See Also

cmatmadd, cmatssub

cmatsadd

complex matrix + scalar addition

Synopsis

```
#include <cmatrix.h>

complex_float *cmatsaddf (complex_float dm *output,
                          const complex_float dm *a,
                          complex_float scalar,
                          int rows, int cols);

complex_double *cmatsadd (complex_double dm *output,
                          const complex_double dm *a,
                          complex_double scalar,
                          int rows, int cols);

complex_long_double *cmatsaddd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          complex_long_double scalar,
                          int rows, int cols);
```

Description

The cmatsadd functions add a complex scalar to each element of the complex input matrix `a[][]` and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The cmatsadd functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8
```

```
complex_double a[ROWS][COLS], *a_p = (double_complex *) (&a);
complex_double c[ROWS][COLS], *res_p = (double_complex *) (&c);
complex_double z;

cmatsadd (res_p, a_p, z, ROWS, COLS);
```

See Also

cmatmadd, cmatssub

cmatsmlt

complex matrix * scalar multiplication

Synopsis

```
#include <cmatrix.h>

complex_float *cmatsmltf (complex_float dm *output,
                          const complex_float dm *a,
                          complex_float scalar
                          int rows, int cols);

complex_double *cmatsmlt (complex_double dm *output,
                          const complex_double dm *a,
                          complex_double scalar,
                          int rows, int cols);

complex_long_double *cmatsmltd (complex_long_double dm *output,
                                const complex_long_double dm *a,
                                complex_long_double scalar,
                                int rows, int cols);
```

Description

The cmatsmlt functions multiply each element of the complex input matrix `a[][]` with a complex scalar, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Complex matrix by scalar multiplication is defined by the following algorithm:

$$\mathrm{Re}(c_{i,j}) = \mathrm{Re}(a_{i,j}) * \mathrm{Re}(scalar) - \mathrm{Im}(a_{i,j}) * \mathrm{Im}(scalar)$$
$$\mathrm{Im}(c_{i,j}) = \mathrm{Re}(a_{i,j}) * \mathrm{Im}(scalar) + \mathrm{Im}(a_{i,j}) * \mathrm{Re}(scalar)$$

where i={0,1,2,..., rows-1}, j={0,1,2,..., cols-1}

Error Conditions

The cmatsmlt functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (double_complex *) (&a);
complex_double c[ROWS][COLS], *res_p = (double_complex *) (&c);
complex_double z;

cmatsmlt (res_p, a_p, z, ROWS, COLS);
```

See Also

cmatmmlt

cmatssub

complex matrix - scalar subtraction

Synopsis

```
#include <cmatrix.h>

complex_float *cmatssubf (complex_float dm *output,
                          const complex_float dm *a,
                          complex_float scalar,
                          int rows, int cols);

complex_double *cmatssub (complex_double dm *output,
                          const complex_double dm *a,
                          complex_double scalar,
                          int rows, int cols);

complex_long_double *cmatssubd (complex_long_double dm *output,
                                const complex_long_double dm *a,
                                complex_long_double scalar,
                                int rows, int cols);
```

Description

The cmatssub functions subtract a complex scalar from each element of the complex input matrix `a[][]` and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The cmatssub functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8
```

```
complex_double a[ROWS][COLS], *a_p = (double_complex *) (&a);
complex_double c[ROWS][COLS], *res_p = (double_complex *) (&c);
complex_double z;

cmatssub (res_p, a_p, z, ROWS, COLS);
```

See Also

cmatmsub, cmatsadd

cmlt

complex multiplication

Synopsis

```
#include <complex.h>

complex_float cmltf (complex_float a, complex_float b);
complex_double cmlt (complex_double a, complex_double b);
complex_long_double cmltd (complex_long_double a,
                           complex_long_double b);
```

Description

The cmlt functions compute the complex multiplication of the complex numbers `a` and `b`, and return the result.

Error Conditions

The cmlt functions do not return any error conditions.

Example

```
#include <complex.h>

complex_float x = {3.0,11.0};
complex_float y = {1.0, 2.0};
complex_float z;

z = cmltf(x,y);     /* z.re = 14.0, z.im = 22.0 */
```

See Also

cadd, cdiv, csub

conj

complex conjugate

Synopsis

```
#include <complex.h>

complex_float conjf (complex_float a);
complex_double conj (complex_double a);
complex_long_double conjd (complex_long_double a);
```

Description

These functions conjugate the complex input `a`, and return the result.

Error Conditions

The conj functions do not return any error conditions.

Example

```
#include <complex.h>

complex_double x = {2.0,8.0};
complex_double z;

z = conj(x);     /* z = (2.0,-8.0) */
```

See Also

No references to this function.

convolve

convolution

Synopsis

```
#include <trans.h>
float *convolve (const float a[], int asize,
                 const float b[], int bsize, float output);
```

Description

This function calculates the convolution of the input vectors `a[]` and `b[]`, and returns the result in the vector `output[]`. The lengths of these vectors are `a[asize]`, `b[bsize]`, and `output[asize+bsize-1]`.

The function returns a pointer to the output vector.

Convolution of two vectors is defined as:

$$c_k = \sum_{j=m}^{n} a_j * b_{(k-j)}$$

where

```
k = {0, 1, ..., asize + bsize - 2}
m = max( 0, k + 1 - bsize)
n = min( k, asize - 1)
```

Error Conditions

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

Example

```
#include <trans.h>
```

```
float input[81];
float response[31];
float output[81 + 31 -1];

convolve(input,81,response,31,output);
```

See Also

crosscorr, ifftN, rfftN

copysign

copy the sign of the floating-point operand (IEEE arithmetic function)

Synopsis

```
#include <math.h>
double copysign (double x, double y);
float copysignf (float x, float y);
long double copysignd (long double x, long double y);
```

Description

The copysign functions copy the sign of the second argument `y` to the first argument `x` without changing either its exponent or mantissa.

The `copysignf` function is a built-in function which is implemented with an `Fn=Fx COPYSIGN Fy` instruction. The `copysign` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

These functions do not return an error code.

Example

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

x = copysign (0.5, -10.0);        /* x = -0.5 */
y = copysignf (-10.0, 0.5f);      /* y = 10.0 */
```

See Also

No references to this function.

cot

cotangent

Synopsis

```
#include <math.h>
double cot (double x);
float cotf (float x);
long double cotd (long double x);
```

Description

The cot functions return the cotangent of their argument. The input is interpreted as radians.

Error Conditions

The input argument `x` for `cotf` must be in the domain [-1.647e6, 1.647e6] and the input argument for `cotd` must be in the domain [-4.21657e8, 4.21657e8]. The functions return zero if `x` is outside their domain.

Example

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

y = cot (x);
v = cotf (w);
```

See Also

tan

crosscoh

cross-coherence

Synopsis

```
#include <stats.h>

float *crosscohf (float dm out[],
                  const float dm x[], const float dm y[],
                  int samples, int lags);

double *crosscoh (double dm out[],
                  const double dm x[], const double dm y[],
                  int samples, int lags);

long double *crosscohd (long double dm out[],
                        const long double dm x[],
                        const long double dm y[],
                        int samples, int lags);
```

Description

The crosscoh functions compute the cross-coherence of two floating-point inputs, `x[]` and `y[]`. The cross-coherence is the cross-correlation minus the product of the mean of `x` and the mean of `y`. The length of the input arrays is given by `samples`. The functions return a pointer to the output array `out[]` of length `lags`.

Error Conditions

The crosscoh functions do not return an error condition.

Example

```
#include <stats.h>
#define SAMPLES 1024
#define LAGS      16
```

```
double excitation[SAMPLES], y[SAMPLES];
double response[LAGS];
int lags = LAGS;

crosscoh (response, excitation, y, SAMPLES, lags);
```

See Also

autocoh, autocorr, crosscorr

crosscorr

cross-correlation

Synopsis

```
#include <stats.h>

float *crosscorrf (float dm out[],
                   const float dm x[], const float dm y[],
                   int samples, int lags);

double *crosscorr (double dm out[],
                   const double dm x[], const double dm y[],
                   int samples, int lags);

long double *crosscorrd (long double dm out[],
                         const long double dm x[],
                         const long double dm y[],
                         int samples, int lags);
```

Description

The crosscorr functions perform a cross-correlation between two signals. The cross-correlation is the sum of the scalar products of the signals in which the signals are displaced in time with respect to one another. The signals to be correlated are given by the input arrays `x[]` and `y[]`. The length of the input arrays is given by `samples`. The functions return a pointer to the output data array `out[]` of length `lags`.

Cross-correlation is used in signal processing applications such as speech analysis.

Error Conditions

The crosscorr functions do not return an error condition.

Example

```
#include <stats.h>
#define SAMPLES 1024
#define LAGS      16

double excitation[SAMPLES], y[SAMPLES];
double response[LAGS];
int lags = LAGS;

crosscorr (response, excitation, y, SAMPLES, lags);
```

See Also

autocoh, autocorr, crosscoh

csub

complex subtraction

Synopsis

```
#include <complex.h>
complex_float csubf(complex_float a, complex_float b);
complex_double csub (complex_double a, complex_double b);
complex_long_double csubd (complex_long_double a,
                           complex_long_double b);
```

Description

The csub functions subtract the two complex values `a` and `b`, and return the result.

Error Conditions

The csub functions do not return any error conditions.

Example

```
#include <complex.h>

complex_float x = {9.0,16.0};
complex_float y = {1.0,-1.0};
complex_float z;

z = csubf(x,y);     /* z.re = 8.0, z.im = 17.0 */
```

See Also

cadd, cdiv, cmlt

cvecdot

complex vector dot product

Synopsis

```
#include <cvector.h>

complex_float cvecdotf (const complex_float dm a[],
                        const complex_float dm b[], int samples);

complex_double cvecdot (const complex_double dm a[],
                        const complex_double dm b[], int samples);

complex_long_double cvecdotd (const complex_long_double dm a[],
                              const complex_long_double dm b[],
                              int samples);
```

Description

The cvecdot functions compute the complex dot product of the complex vectors `a[]` and `b[]`, which are `samples` in size. The scalar result is returned by the function.

The algorithm for a complex dot product is given by:

$$\mathrm{Re}(c_i) = \sum_{l=0}^{n-1} \mathrm{Re}(a_i) * \mathrm{Re}(b_i) - \mathrm{Im}(a_i) * \mathrm{Im}(b_i)$$

$$\mathrm{Im}(c_i) = \sum_{l=0}^{n-1} \mathrm{Re}(a_i) * \mathrm{Im}(b_i) + \mathrm{Im}(a_i) * \mathrm{Re}(b_i)$$

Error Conditions

The cvecdot functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float x[N], y[N];
complex_float answer;

answer = cvecdotf (x, y, N);
```

See Also

vecdot

cvecsadd

complex vector + scalar addition

Synopsis

```
#include <cvector.h>

complex_float *cvecsaddf (const complex_float dm a[],
                          complex_float scalar,
                          complex_float dm output[], int samples);

complex_double *cvecsadd (const complex_double dm a[],
                          complex_double scalar,
                          complex_double dm output[], int samples);

complex_long_double *cvecsaddd (const complex_long_double dm a[],
                                complex_long_double scalar,
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecsadd functions compute the sum of each element of the complex vector `a[]`, added to the complex scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The cvecsadd functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input[N], result[N];
complex_float x;
```

```
cvecsaddf (input, x, result, N);
```

See Also

vecvadd

cvecsmlt

complex vector * scalar multiplication

Synopsis

```
#include <cvector.h>

complex_float *cvecsmltf (const complex_float dm a[],
                          complex_float scalar,
                          complex_float dm output[], int samples);

complex_double *cvecsmlt (const complex_double dm a[],
                          complex_double scalar,
                          complex_double dm output[], int samples);

complex_long_double *cvecsmltd (const complex_long_double dm a[],
                                complex_long_double scalar,
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecsmlt functions compute the product of each element of the complex vector `a[]`, multiplied by the complex scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Complex vector by scalar multiplication is given by the formula:

$$\mathrm{Re}(c_i) = \mathrm{Re}(a_i)*\mathrm{Re}(scalar) - \mathrm{Im}(a_i)*\mathrm{Im}(scalar)$$

$$\mathrm{Im}(c_i) = \mathrm{Re}(a_i)*\mathrm{Im}(scalar) + \mathrm{Im}(a_i)*\mathrm{Re}(scalar)$$

where: $i=\{0,1,2,\ldots,samples-1\}$

Error Conditions

The cvecsmlt functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input[N], result[N];
complex_float x;

cvecsmltf (input, x, result, N);
```

See Also

vecvmlt

cvecssub

complex vector - scalar subtraction

Synopsis

```
#include <cvector.h>

complex_float *cvecssubf (const complex_float dm a[],
                          complex_float scalar,
                          complex_float dm output[], int samples);

complex_double *cvecssub (const complex_double dm a[],
                          complex_double scalar,
                          complex_double dm output[], int samples);

complex_long_double *cvecssubd (const complex_long_double dm a[],
                                complex_long_double scalar,
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecssub functions compute the difference of each element of the complex vector `a[]`, minus the complex scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The cvecssub functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input[N], result[N];
complex_float x;
```

```
cvecssubf (input, x, result, N);
```

See Also

vecvsub

cvecvadd

complex vector + vector addition

Synopsis

```
#include <cvector.h>

complex_float *cvecvaddf (const complex_float dm a[],
                          const complex_float dm b[],
                          complex_float dm output[], int samples);

complex_double *cvecvadd (const complex_double dm a[],
                          const complex_double dm b[],
                          complex_double dm output[], int samples);

complex_long_double *cvecvaddd (const complex_long_double dm a[],
                                const complex_long_double dm b[],
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecvadd functions compute the sum of each of the elements of the complex vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The cvecvadd functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input_1[N];
complex_float input_2[N], result[N];
```

```
cvecvaddf (input_1, input_2, result, N);
```

See Also

vecvadd

cvecvmlt

complex vector * vector multiplication

Synopsis

```
#include <cvector.h>

complex_float *cvecvmltf (const complex_float dm a[],
                          const complex_float dm b[],
                          complex_float dm output[], int samples);

complex_double *cvecvmlt (const complex_double dm a[],
                          const complex_double dm b[],
                          complex_double dm output[], int samples);

complex_long_double *cvecvmltd (const complex_long_double dm a[],
                                const complex_long_double dm b[],
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecvmlt functions compute the product of each of the elements of the complex vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size.The functions return a pointer to the output vector.

Complex vector multiplication is given by the formula:

$$\mathrm{Re}(c_i) = \mathrm{Re}(a_i)*\mathrm{Re}(b_i) - \mathrm{Im}(a_i)*\mathrm{Im}(b_i)$$
$$\mathrm{Im}(c_i) = \mathrm{Re}(a_i)*\mathrm{Im}(b_i) + \mathrm{Im}(a_i)*\mathrm{Re}(b_i)$$

where `i={0,1,2,...,samples-1}`

Error Conditions

The cvecvmlt functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input_1[N];
complex_float input_2[N], result[N];

cvecvmltf (input_1, input_2, result, N);
```

See Also

vecvmlt

cvecvsub

complex vector - vector subtraction

Synopsis

```
#include <cvector.h>

complex_float *cvecvsubf (const complex_float dm a[],
                          const complex_float dm b[],
                          complex_float dm output[], int samples);

complex_double *cvecvsub (const complex_double dm a[],
                          const complex_double dm b[],
                          complex_double dm output[], int samples);

complex_long_double *cvecvsubd (const complex_long_double dm a[],
                                const complex_long_double dm b[],
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecvsub functions compute the difference of each of the elements of the complex vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The cvecvsub functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input_1[N];
complex_float input_2[N], result[N];
```

```
cvecvsubf (input_1, input_2, result, N);
```

See Also

vecvsub

favg

mean of two values

Synopsis

```
#include <math.h>

double favg (double x, double y);
float favgf (float x, float y);
long double favgd (long double x, long double y);
```

Description

The favg functions return the mean of their two arguments.

The `favgf` function is a built-in function which is implemented with an `Fn=(Fx+Fy)/2` instruction. The `favg` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The favg functions do not return an error code.

Example

```
#include <math.h>

float x;
x = favgf (10.0f, 8.0f);     /* returns 9.0f */
```

See Also

avg, lavg

fclip

clip

Synopsis

```
#include <math.h>

double fclip (double x, double y);
float fclipf (float x, float y);
long double fclipd (long double x, long double y);
```

Description

The fclip functions return the 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 an `Fn=CLIP Fx BY Fy` instruction. The `fclip` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The fclip functions do not return an error code.

Example

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

y = fclipf (5.1f, 8.0f);     /* returns 5.1f */
```

See Also

clip, lclip

fir

finite impulse response (FIR) filter

Synopsis

```
#include <filters.h>

float fir (float sample,
           const float pm coeffs[],
           float dm state[],
           int taps);
```

Description

The `fir` function implements a finite impulse response (FIR) filter defined by the coefficients and delay line that are supplied in the call of `fir`. The function produces the filtered response of its input data. This FIR filter is structured as a sum of products. The characteristics of the filter (passband, stop band, etc.) are dependent on the coefficient values and the number of taps supplied by the calling program.

The floating-point input to the filter is `sample`. The integer `taps` indicates the length of the filter, which is also the length of the array `coeffs`. The `coeffs` array holds one FIR filter coefficient per element. The coefficients should be stored in reverse order with `coeffs[0]` containing the last `(taps-1)` coefficient and `coeffs[taps-1]` containing the first coefficient. The `coeffs` array must be located in program memory data space so that the single-cycle dual-memory fetch of the processor can be used.

The `state` array contains a pointer to the delay line as its first element, followed by the delay line values. The length of the `state` array is therefore one greater than the number of taps. Each filter has its own `state` array, which should not be modified by the calling program, only by the `fir` function. The `state` array should be initialized to zeros before the `fir` function is called for the first time.

Error Conditions

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

Example

```
#include <filters.h>
#define TAPS 10

float y;
float pm coeffs[TAPS];    /* coeffs array must be initialized */
                          /* and in PM memory                 */
float state[TAPS+1];
int i;

for (i = 0; i < TAPS+1; i++)
    state[i] = 0;         /* initialize state array           */

y = fir (0.775, coeffs, state, TAPS);
                          /* y holds the filtered output      */
```

See Also

biquad, iir

fmax

maximum

Synopsis

```
#include <math.h>

double fmax (double x, double y);
float fmaxf (float x, float y);
long double fmaxd (long double x, long double y);
```

Description

The fmax functions return the larger of their two arguments.

The `fmaxf` function is a built-in function which is implemented with an `Fn=MAX(Fx,Fy)` instruction. The `fmax` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The fmax functions do not return an error code.

Example

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

y = fmaxf (5.1f, 8.0f);      /* returns 8.0f */
```

See Also

fmin, lmax, lmin, max, min

fmin

minimum

Synopsis

```
#include <math.h>

double fmin (double x, double y);
float fminf (float x, float y);
long double fmind (long double x, long double y);
```

Description

The fmin functions return the smaller of their two arguments.

The `fminf` function is a built-in function which is implemented with an `Fn=MIN(Fx,Fy)` instruction. The `fmin` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The fmin functions do not return an error code.

Example

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

y = fminf (5.1f, 8.0f);      /* returns 5.1f */
```

See Also

fmax, lmax, lmin, max, min

gen_bartlett

generate bartlett window

Synopsis

```
#include <window.h>
void gen_bartlett(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                         */);
```

Description

The `gen_bartlett` function generates a vector containing the Bartlett window. The length is specified by parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

The Bartlett window is similar to the Triangle window (see "gen_triangle" on page 4-90) but has the following different properties

- The Bartlett window always returns a window with two zeros on either end of the sequence. Therefore, for odd `n`, the center section of a `N+2` Bartlett window equals a `N` Triangle window.
- For even `n`, the Bartlett window is the convolution of two rectangular sequences. There is no standard definition for the Triangle window for even `n`; the slopes of the Triangle window are slightly steeper than those of the Bartlett window.

Algorithm

$$w[n] = 1 - \left| \frac{n - \frac{N-1}{2}}{\frac{N-1}{2}} \right|$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_blackman

generate blackman window

Synopsis

```
#include <window.h>
void gen_blackman(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_blackman` 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, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_gaussian

generate gaussian window

Synopsis

```
#include <window.h>
void gen_gaussian(
float dm w[], /* Window vector                                    */
float alpha,  /* Gaussian alpha parameter                         */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_gaussian` 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, ..., N-1} and α is an input parameter

Domain

$a > 0$; $N > 0$; $\alpha > 0.0$

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_hamming

generate hamming window

Synopsis

```
#include <window.h>
void gen_hamming(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_hamming` function generates a vector containing the Hamming window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Algorithm

$$w[n] = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right)$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_hanning

generate hanning window

Synopsis

```
#include <window.h>
void gen_hanning(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_hanning` function generates a vector containing the Hanning window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. This window is also known as the Cosine window.

Algorithm

$$w[n] = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right)$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_harris

generate harris window

Synopsis

```
#include <window.h>
void gen_harris(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_harris` 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, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_kaiser

generate kaiser window

Synopsis

```
#include <window.h>
void gen_kaiser(
float dm w[], /* Window vector                                  */
float beta,   /* Kaiser beta parameter                         */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                       */);
```

Description

The `gen_kaiser` 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, ..., N-1}, α= (N - 1) /2, and $I_0(\beta)$ represents the zeroth-order modified Bessel function of the first kind.

Domain

a > 0; N > 0; β > 0.0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_rectangular, gen_triangle, gen_vonhann

gen_rectangular

generate rectangular window

Synopsis

```
#include <window.h>
void gen_rectangular(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_rectangular` function generates a vector containing the Rectangular window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Algorithm

w[n] = 1

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_triangle, gen_vonhann

gen_triangle

generate triangle window

Synopsis

```
#include <window.h>
void gen_triangle(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_triangle` 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*a`.

Refer to the Bartlett window (described on page 4-80) regarding the relationship between it and the Triangle window.

Algorithm

For even `n`, the following equation applies:

$$w[n] = \begin{cases} \dfrac{2n+1}{N} & n < N/2 \\ \dfrac{2N-2n-1}{N} & n > N/2 \end{cases}$$

where n = {0, 1, 2, ..., N-1}

For odd `n`, the following equation applies:

$$w[n] = \begin{cases} \dfrac{2n+2}{N+1} & n < N/2 \\ \dfrac{2N-2n}{N+1} & n > N/2 \end{cases}$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_vonhann

gen_vonhann

generate von hann window

Synopsis

```
#include <window.h>
void gen_vonhann(
float dm 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 4-85).

Error Conditions

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

See Also

gen_hanning

histogram

histogram

Synopsis

```
#include <stats.h>

int *histogram (int out[],
                const int in[],
                int out_len,
                int samples,
                int bin_size);
```

Description

The `histogram` function computes a scaled-integer histogram of its input array. The `bin_size` parameter is used to adjust the width of each individual bin in the output array. For example, a `bin_size` of 5 indicates that the first location of the output array holds the number of occurrences of a `0`, `1`, `2`, `3` or `4`.

The output array is first zeroed by the function, and then each sample in the input array is multiplied by `1/bin_size` and truncated. The appropriate bin in the output array is incremented. This function returns a pointer to the output array.

For maximum performance, this function does not perform out of bounds checking. Therefore, all values within the input array must be within range (that is, between 0 and `bin_size * out_len`).

Error Conditions

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

Example

```
#include <stats.h>
#define SAMPLES 1024

int length = 2048;
int excitation[SAMPLES], response[2048];
histogram (response, excitation, length, SAMPLES, 5);
```

See Also

mean, var

idle

execute ADSP-21xxx processor's IDLE instruction

Synopsis

```
#include <21020.h>
#include <21060.h>
#include <21065l.h>
void idle (void);
```

Description

The `idle` function invokes the processor's `idle` instruction once and returns. The `idle` instruction causes the processor to stop and respond only to interrupts. For a complete description of the `idle` instruction, please refer to the appropriate hardware reference manual for the target processor.

In earlier releases of the VisualDSP++ software (prior to release 2.1), the `idle` function repeatedly executed the `idle` instruction. This function has been changed to give you more control over the amount of time spent in the `idle` state.

Error Conditions

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

Example

```
#include <21060.h>
idle ();
```

See Also

interrupt, signal

ifftN

N-point inverse FFT

Synopsis

```
#include <trans.h>

float *ifft65536 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft32768 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft16384 (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft8192  (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft4096  (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft2048  (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft1024  (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft512   (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);
```

```
float *ifft256   (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft128   (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft64    (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft32    (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft16    (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);

float *ifft8     (const float dm real_input[],
                  const float dm imag_input[],
                  float dm real_output[], float dm imag_output[]);
```

Description

Each of these 14 `ifftN` functions computes the N-point radix-2 inverse fast Fourier transform (IFFT) of its floating-point input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are 14 distinct functions in this set. They all perform the same function with the same type and number of arguments. The only difference between them is the size of the arrays on which they operate. To call a particular function, substitute the number of points for N. For example,

```
ifft8 (r_inp, i_inp, r_outp, i_outp);
```

The input to `ifftN` are two floating-point arrays of N points. The array `real_input` contains the real components of the inverse FFT input and the array `imag_input` contains the imaginary components.

If there are fewer than N actual data points, you must pad the arrays with zeros to make N samples. However, better results occur with less zero padding. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data.

The time-domain signal generated by the `ifftN` functions is stored in the arrays `real_output` and `imag_output`. The array `real_output` contains the real component of the complex output signal while the array `imag_output` contains the imaginary component. The output is scaled by `N`, the number of points in the inverse FFT. The functions return a pointer to the `real_output` array.

If the input data can be overwritten, then the `ifftN` functions allow the array `real_input` to share the same memory as the array `real_output`, and `imag_input` to share the same memory as `imag_output`. This would improve memory usage, but at the cost of some run-time performance.

Error Conditions

The `ifftN` functions do not return error conditions.

Example

```
#include <trans.h>
#define N 2048

float real_input[N], imag_input[N];
float real_output[N], imag_output[N];

   /* Real input arrays filled from a previous xfft2048() or
   other source */
ifft2048 (real_input, imag_input, real_output, imag_output);
   /* Arrays are filled with FFT data */
```

See Also

cfftN, rfftN

iir

infinite impulse response (IIR) filter

Synopsis

```
#include <filters.h>

float iir (float sample,
           const float pm a_coeffs[],
           const float pm b_coeffs[],
           float dm state[],
           int taps);
```

Description

The `iir` function implements an infinite impulse response (IIR) filter based on the Oppenheim and Schafer direct form II. The function returns the filtered response of the input data `sample`. The characteristics of the filter are dependent upon a set of coefficients, a delay line, and the length of the filter. The length of filter is specified by the argument `taps`.

The set of IIR filter coefficients is composed of a-coefficients and b-coefficients. The a_0 coefficient is assumed to be `1.0`, and the remaining a-coefficients should be scaled accordingly and stored in the array `a_coeffs` in reverse order. The length of the `a_coeffs` array is `taps` and therefore `a_coeffs[0]` should contain a_{taps}, and `a_coeffs[taps-1]` should contain a_1.

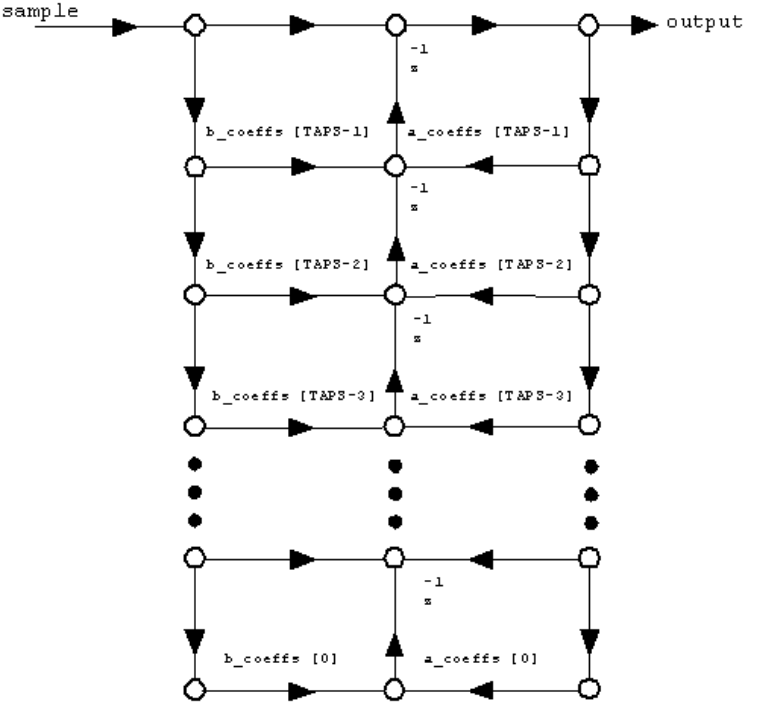
The b-coefficients are stored in the array `b_coeffs`, also in reverse order. The length of the `b_coeffs` is `taps+1`, and so `b_coeffs[0]` contains b_{taps} and `b_coeffs[taps]` contains b_0.

Both the `a_coeffs` and `b_coeffs` arrays must be located in program memory (PM) so that the single-cycle dual-memory fetch of the processor can be used.

When importing coefficients from a filter design tool that employs a transposed direct form II, the a_1 and a_2 coefficients have to be negated. For example, if a filter design tool returns `A = [1.0, 0.2, -0.9]`, then the a-coefficients first have to be inverted to `A = [1.0, -0.2, 0.9]`.

Each filter should have its own delay line which the function maintains in the array `state`. The array should be initialized to zero before calling the function for the first time and should not otherwise be modified by the calling program. The length of the `state` array should be `taps+1` as the function uses the array to store a pointer to the current delay line.

The flow graph (below) corresponds to the `iir()` routine as part of the DSP run-time library.



The b_coeffs array should equal TAPS+1
The a_coeffs array should equal TAPS

The `biquad` function should be used instead of the `iir` function if a multi-stage filter is required

Error Conditions

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

Example

```
#include <filters.h>
#define NSAMPLES  256
#define TAPS       10

float input[NSAMPLES];
float output[NSAMPLES];

float pm a_coeffs[TAPS];
float pm b_coeffs[TAPS+1];
float state[TAPS+1];
int i;

for (i = 0; i < TAPS+1; i++)
    state[i] = 0;             /* initialize state array */

for (i = 0; i < NSAMPLES; i++) {
    output[i] = iir (input[i], a_coeffs, b_coeffs, state, TAPS);
```

See Also

biquad, fir

matinv

real matrix inversion

Synopsis

```
#include <matrix.h>

float *matinvf (float dm *output,
                const float dm *input, int samples);

double *matinv (double dm *output,
                const double dm *input, int samples);

long double *matinvd (long double dm *output,
                      const long double dm *input, int samples);
```

Description

The matinv functions employ Gauss-Jordan elimination with full pivoting to compute the inverse of the input matrix `input` and store the result in the matrix `output`. The dimensions of the matrices `input` and `output` are `[n][n]`. The functions return a pointer to the output matrix.

Error Conditions

If no inverse exists for the input matrix, the functions return a null pointer.

Example

```
#include <matrix.h>
#define N 8

double a[N][N};
double a_inv[N]{N};

matinv ((double *)(a_inv),(double *)(a),N);
```

See Also

No references available.

matmadd

real matrix + matrix addition

Synopsis

```
#include <matrix.h>

float *matmaddf (float dm *output,
                 const float dm *a,
                 const float dm *b, int rows, int cols);

double *matmadd (double dm *output,
                 const double dm *a,
                 const double dm *b, int rows, int cols);

long double *matmaddd (long double dm *output,
                 const long double dm *a,
                 const long double dm *b, int rows, int cols);

float *matadd  (float dm *output,
                 const float dm *a,
                 const float dm *b, int rows, int cols);
```

Description

The matmadd functions perform a matrix addition of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`.

The functions return a pointer to the output matrix.

The `matadd` function is equivalent to `matmaddf` and is provided for compatibility with previous versions of VisualDSP++.

Error Conditions

The matmadd functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input_1[ROWS][COLS], *a_p = (double *) (&input_1);
double input_2[ROWS][COLS], *b_p = (double *) (&input_2);
double result[ROWS][COLS], *res_p = (double *) (&result);

matmadd (res_p, a_p, b_p, ROWS, COLS);
```

See Also

matmsub

matmmlt

real matrix * matrix multiplication

Synopsis

```
#include <matrix.h>

float *matmmltf (float dm *output,
                 const float dm *a,
                 const float dm *b,
                 int a_rows, int a_cols, b_cols);

double *matmmlt (double dm *output,
                 const double dm *a,
                 const double dm *b,
                 int a_rows, int a_cols, b_cols);

long double *matmmltd (long double dm *output,
                 const long double dm *a,
                 const long double dm *b,
                 int a_rows, int a_cols, b_cols);

float *matmul   (float dm *output,
                 const float dm *a,
                 const float dm *b,
                 int a_rows, int a_cols, b_cols);
```

Description

The matmmlt functions perform a matrix multiplication of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[a_rows][a_cols]`, `b[a_cols][b_cols]`, and `output[a_rows][b_cols]`. The functions return a pointer to the output matrix.

Matrix multiplication is defined by the following algorithm:

$$c_{i,j} = \sum_{l=0}^{a_cols-1} a_{i,l} * b_{l,j}$$

where `i={0,1,2,...,a_rows-1}, j={0,1,2,...,b_cols-1}`

The `matmul` function is equivalent to `matmmltf` and is provided for compatibility with previous versions of VisualDSP++.

Error Conditions

The matmmlt functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS_1 4
#define COLS_1 8
#define COLS_2 2

double input_1[ROWS_1][COLS_1], *a_p = (double *) (&input_1);
double input_2[COLS_1][COLS_2], *b_p = (double *) (&input_2);
double result[ROWS_1][COLS_2], *res_p = (double *) (&result);

matmmlt (res_p, a_p, b_p, ROWS_1, COLS_1, COLS_2);
```

See Also

matsmlt

matmsub

real matrix - matrix subtraction

Synopsis

```
#include <matrix.h>

float *matmsubf (float dm *output,
                 const float dm *a,
                 const float dm *b, int rows, int cols);

double *matmsub (double dm *output,
                 const double dm *a,
                 const double dm *b, int rows, int cols);

long double *matmsubd (long double dm *output,
                 const long double dm *a,
                 const long double dm *b, int rows, int cols);

float *matsub   (float dm *output,
                 const float dm *a,
                 const float dm *b, int rows, int cols);
```

Description

The matmsub functions perform a matrix subtraction of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`.The functions return a pointer to the output matrix.

The `matsub` function is equivalent to `matmsubf` and is provided for compatibility with previous versions of VisualDSP++.

Error Conditions

The matmsub functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input_1[ROWS][COLS], *a_p = (double *) (&input_1);
double input_2[ROWS][COLS], *b_p = (double *) (&input_2);
double result[ROWS][COLS], *res_p = (double *) (&result);

matmsub (res_p, a_p, b_p, ROWS, COLS);
```

See Also

matmadd

matsadd

real matrix + scalar addition

Synopsis

```
#include <matrix.h>

float *matsaddf (float dm *output, const float dm *a,
                 float scalar, int rows, int cols);

double *matsadd (double dm *output, const double dm *a
                 double scalar, int rows, int cols);

long double *matsaddd (long double dm *output,
                       const long double dm *a,
                       long double scalar, int rows, int cols);
```

Description

The matsadd functions add a scalar to each element of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The matsadd functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input[ROWS][COLS], *a_p = (double *) (&input);
double result[ROWS][COLS], *res_p = (double *) (&result);
double x;
```

```
matsadd (res_p, a_p, x, ROWS, COLS);
```

See Also

matmadd, matssub

matsmlt

real matrix * scalar multiplication

Synopsis

```
#include <matrix.h>

float *matsmltf (float dm *output, const float dm *a,
                 float scalar, int rows, int cols);

double *matsmlt (double dm *output, const double dm *a
                 double scalar, int rows, int cols);

long double *matsmltd (long double dm *output,
                       const long double dm *a,
                       long double scalar, int rows, int cols);

float *matscalmult (float dm *output, const float dm *a,
                    float scalar, int rows, int cols);
```

Description

The matsmlt functions multiply a scalar with each element of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`.

The functions return a pointer to the output matrix.

The `matscalmult` function is equivalent to `matsmltf` and is provided for compatibility with previous versions of VisualDSP++.

Error Conditions

The matsmlt functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input[ROWS][COLS], *a_p = (double *) (&input);
double result[ROWS][COLS], *res_p = (double *) (&result);
double x;

matsmlt (res_p, a_p, x, ROWS, COLS);
```

See Also

matmmlt

matssub

real matrix - scalar subtraction

Synopsis

```
#include <matrix.h>

float *matssubf (float dm *output, const float dm *a,
                 float scalar, int rows, int cols);

double *matssub (double dm *output, const double dm *a
                 double scalar, int rows, int cols);

long double *matssubd (long_double dm *output,
                       const long double dm *a,
                       long double scalar, int rows, int cols);
```

Description

The matssub functions subtract a scalar from each element of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The matssub functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input[ROWS][COLS], *a_p = (double *) (&input);
double result[ROWS][COLS], *res_p = (double *) (&result);
double x;
```

```
matssub (res_p, a_p, x, ROWS, COLS);
```

See Also

matmsub, matsadd

mean

mean

Synopsis

```
#include <stats.h>

float meanf (const float in[], int length);
double mean (const double in[], int length);
long double meand (const long double in[], int length);
```

Description

These functions return the mean of the input array `in[]`. The length of the input array is `length`.

Error Conditions

The mean functions do not return an error condition.

Example

```
#include <stats.h>
#define SIZE 256

double data[SIZE];
double result;

result = mean (data, SIZE);
```

See Also

var

mu_compress

µ-law compression

Synopsis

```
#include <comm.h>
int mu_compress (int x);
```

Description

The `mu_compress` function takes a linear 14-bit signed speech sample and compresses it according to ITU recommendation G.711. The value returned is an 8-bit sample that can be sent directly to a µ-law codec.

Error Conditions

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

Example

```
#include <comm.h>
int linear, compressed;

compressed = mu_compress (linear);
```

See Also

a_compress, mu_expand

mu_expand

µ-law expansion

Synopsis

```
#include <comm.h>
int mu_expand (int x);
```

Description

The `mu_expand` function takes an 8-bit compressed speech sample and expands it according to ITU recommendation G.711 (µ-law definition). The value returned is a linear 14-bit signed sample.

Error Conditions

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

Example

```
#include <comm.h>
int compressed_sample, expanded;

expanded = mu_expand (compressed_sample);
```

See Also

a_expand, mu_compress

norm

normalization

Synopsis

```
#include <complex.h>

complex_float normf (complex_float a);
complex_double norm (complex_double a);
complex_long_double normfd(complex_long_double a);
```

Description

These functions normalize the complex input `a` and return the result. Normalization of a complex number is defined as:

$$\mathrm{Re}(c) = \frac{\mathrm{Re}(a)}{\sqrt{\mathrm{Re}^2(a) + \mathrm{Im}^2(a)}}$$

$$\mathrm{Im}(c) = \frac{\mathrm{Im}(a)}{\sqrt{\mathrm{Re}^2(a) + \mathrm{Im}^2(a)}}$$

Error Conditions

The norm functions return zero if `cabs(a)` is equal to zero.

Example

```
#include <complex.h>

complex_double x = {2.0,-4.0};
complex_double z;

z = norm(x);      /* z = (0.4472,-0.8944) */
```

See Also

No references to this function.

polar

construct from polar coordinates

Synopsis

```
#include <complex.h>

complex_float polarf (complex_float mag, complex_float phase);
complex_double polar (complex_double mag, complex_double phase);
complex_long_double polard (complex_long_double mag,
                            complex_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.

The algorithm for transforming a polar coordinate into a Cartesian coordinate is:

$\mathrm{Re}(c) = mag * \cos(phase)$
$\mathrm{Im}(c) = mag * \sin(phase)$

Error Conditions

The polar functions do not return any error conditions.

Example

```
#include <complex.h>
#define PI 3.14159265

float magnitude = 2.0;
float phase = PI;
```

```
complex_float z;

z = polarf (magnitude,phase);      /* z.re = -2.0, z.im = 0.0 */
```

See Also

arg, cartesian

poll_flag_in

test input flag

Synopsis

```
#include <21060.h>
#include <21065l.h>
int poll_flag_in (int flag, int mode);
```

Description

The `poll_flag_in` function tests the specified flag (0, 1, 2, 3) for the specified transition (0=low to high, 1=high to low, 2=flag high, 3=flag low, 4=any transition, 5=read flag). The function returns a zero *after* the specified transition has occurred in modes 0-3. In Mode 4, it returns the state of the flag after the transition. In Mode 5, it returns the value of the flag without waiting.

Table 4-5. poll_flag_in Macros and Values

Flag Macro	Value	Mode Macro	Value
READ_FLAG0	0	FLAG_IN_LO_TO_HI	0
READ_FLAG1	1	FLAG_IN_HI_TO_LOW	1
READ_FLAG2	2	FLAG_IN_HI	2
READ_FLAG3	3	FLAG_IN_LOW	3
READ_FLAG3	3	FLAG_IN_TRANSITION	4
READ_FLAG3	3	RETURN_FLAG_STATE	5

This function assumes that the flag direction in the `MODE2` register is already set as an input (the default state at reset).

Error Conditions

The `poll_flag_in` function returns a negative value for an invalid flag or transition mode.

Example

```
#include <21060.h>
poll_flag_in (0, 3);
          /* return zero after transition has occurred */
```

See Also

interrupt, set_flag

rfftN

N-point real input FFT

Synopsis

```
#include <trans.h>

float *rfft65536 (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft32768 (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft16384 (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft8192  (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft4096  (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft2048  (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft1024  (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft256   (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft128   (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft64    (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft32    (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);
```

```
float *rfft16    (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);

float *rfft8     (const float dm real_input[],
                  float dm real_output[], float dm imag_output[]);
```

Description

Each of these fourteen `rfftN` functions are similar to the `cfftN` functions, except that they only take real inputs. They compute the N-point radix-2 fast Fourier transform (RFFT) of their floating-point input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are fourteen distinct functions in this set. They all perform the same function with same type and number of arguments. Their only difference is the size of the arrays on which they operate.

Call a particular function by substituting the number of points for N; for example,

```
rfft8 (r_inp, r_outp, i_outp);
```

The input to `rfftN` is a floating-point array of N points. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. However, better results occur with less zero padding. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data.

If the input data can be overwritten, then the `rfftN` functions allow the array `real_input` to share the same memory as the array `imag_output`. This would improve memory usage with only a minimal run-time penalty.

The `rfftN` functions return a pointer to the `real_output` array.

Error Conditions

The `rfftN` functions do not return any error conditions.

Example

```
#include <trans.h>
#define N 2048

float real_input[N];
float real_output[N], imag_output[N];
   /* Real input array fills from a converter or other source */

rfft2048 (real_input, real_output, imag_output);
   /* Arrays are filled with FFT data */
```

See Also

cfftN, ifftN

rms

root mean square

Synopsis

```
#include <stats.h>

float rmsf (const float in[], int length);
double rms (const double in[], int length);
long double rmsd (const long double in[], int length);
```

Description

These functions return the square root of the mean of the square of the input array `in[]`. The length of the input array is `length`.

Error Conditions

The rms functions do not return an error condition.

Example

```
#include <stats.h>
#define SIZE 256

double data[SIZE];
double result;

result = rms (data, SIZE);
```

See Also

mean, var

rsqrt

reciprocal square root

Synopsis

```
#include <math.h>
double rsqrt (double x);
float rsqrtf (float x);
long double rsqrtd (long double x);
```

Description

The rsqrt functions return the reciprocal positive square root of their argument.

Error Conditions

The rsqrt functions return zero for a negative input.

Example

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

y = rsqrt (2.0);      /* y = 0.707 */
```

See Also

sqrt

set_flag

set ADSP-21xxx processor flags

Synopsis

```
#include <21060.h>
#include <21065l.h>
int set_flag (int flag, int mode);
```

Description

The `set_flag` function is used to set the ADSP-21xxx processor flags to the desired output value.

The function accepts as input a flag number [0-3] and a mode. The mode can be specified as a macro (defined in `21060.h`) or a value [0-3].

Table 4-6. Flag Function Macros and Values

Flag Macro	Value	Mode Macro	Value
SET_FLAG0	0	SET_FLAG	0
SET_FLAG1	1	CLR_FLAG	1
SET_FLAG2	2	TGL_FLAG	2
SET_FLAG3	3	TST_FLAG	3

In addition to setting the flag to the specified value, the function also sets the `MODE2` register to specify that the flag is used for output, not input.

If the `TST_FLAG` macro (or a 3) is specified as the mode, the current value (0 or 1) of the flag is returned as the result of the function.

The `set_flag` function returns a zero upon success (except as noted in the previous paragraph).

Error Conditions

The `set_flag` function returns a non-zero for an error.

Example

```
#include <21060.h>
set_flag (SET_FLAG0, CLR_FLAG);
set_flag (SET_FLAG0, SET_FLAG);
```

See Also

poll_flag_in

set_semaphore

set bus lock semaphore

Synopsis

```
#include <21060.h>
#include <21065l.h>
int set_semaphore (
    void dm *semaphore, int set_value, int timeout);
```

Description

The `set_semaphore` function is used to control bus lock in multiprocessor ADSP-21xxx systems.

- -1 is returned if the bus is locked and the bus lock timeout exceeded.
- 0 is returned if the bus is not locked and a semaphore set.

Error Conditions

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

See Also

No references to this function.

timer_off

disable ADSP-21xxx processor timer

Synopsis

```
#include <21060.h>
unsigned int timer_off (void);
```

Description

The `timer_off` function disables the ADSP-21xxx timer and returns the current value of the `TCOUNT` register.

Error Conditions

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

Example

```
#include <21060.h>
unsigned int hold_tcount;
hold_tcount = timer_off ();
              /* hold_tcount contains value of TCOUNT */
              /* register AFTER timer has stopped     */
```

See Also

timer_on, timer_set

The `timer_off` function is not available for the ADSP-21065L chip. Refer to timer0_off, timer1_off to disable the ADSP-21065L programmable timers.

The function is supplied only as an inlined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_off` must include the `21060.h` header file.

timer0_off, timer1_off

disable ADSP-21065L processor timers

Synopsis

```
#include <21065l.h>
unsigned int timer0_off (void);
unsigned int timer1_off (void);
```

Description

The `timer0_off` and `timer1_off` functions disable the ADSP-21065L programmable timers and return the current value of the `TCOUNT0` and `TCOUNT1` registers, respectively.

Error Conditions

The `timer0_off` and `timer1_off` functions do not return an error condition.

Example

```
#include <21065l.h>
unsigned int hold_tcount;
hold_tcount = timer0_off ();
              /* hold_tcount contains value of TCOUNT0 */
              /* register AFTER timer 0 has stopped    */
```

See Also

timer0_on, timer1_on, timer0_set, timer1_set

The functions are supplied only as inlined procedures; that is, the compiler substitutes the appropriate statements for any reference to the procedures. Therefore, any source that references either `timer0_off` or `timer1_off` must include the `21065l.h` header file.

timer_on

enable ADSP-21xxx processor timer

Synopsis

```
#include <21060.h>
unsigned int timer_on (void);
```

Description

The `timer_on` function enables the ADSP-21xxx timer and returns the current value of the `TCOUNT` register.

Error Conditions

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

Example

```
#include <21060.h>
unsigned int hold_tcount;
hold_tcount = timer_on ();
              /* hold_tcount contains value of TCOUNT */
              /* register when timer starts           */
```

See Also

timer_off, timer_set

The `timer_on` function is not available for the ADSP-21065L chip. Refer to "timer0_on, timer1_on" on page 4-134 to enable the ADSP-21065L programmable timers.

The function is supplied only as an inlined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_on` must include the `21060.h` header file.

timer0_on, timer1_on

enable ADSP-21065L processor timers

Synopsis

```
#include <21065l.h>
unsigned int timer0_on (void);
unsigned int timer1_on (void);
```

Description

The `timer0_on` and `timer1_on` functions enable the ADSP-21065L programmable timers and return the current value of the `TCOUNT0` and `TCOUNT1` registers, respectively.

Error Conditions

The `timer0_on` and `timer1_on` functions do not return an error condition.

Example

```
#include <21065l.h>
unsigned int hold_tcount;
hold_tcount = timer0_on ();
              /* hold_tcount contains value of TCOUNT0 */
              /* register when timer 0 starts          */
```

See Also

timer0_off, timer1_off, timer0_set, timer1_set

The functions are supplied only as inlined procedures; that is, the compiler substitutes the appropriate statements for any reference to the procedures. Therefore, any source that references either `timer0_on` or `timer1_on` must include the `21065l.h` header file.

timer_set

initialize ADSP-21xxx processor timer

Synopsis

```
#include <21060.h>
int timer_set (unsigned int tperiod,
               unsigned int tcount);
```

Description

The `timer_set` function sets the ADSP-21xxx timer registers `TPERIOD` and `TCOUNT`. The function returns a 1 if the timer is enabled, or a zero if the timer is disabled.

Each interrupt call takes approximately 50 cycles on entrance and 50 cycles on return. If `TPERIOD` and `TCOUNT` registers are set too low, you may incur an initializing overhead that could create an infinite loop.

Error Conditions

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

Example

```
#include <21060.h>
if (timer_set (1000, 1000) != 1)
    timer_on ();     /* enable timer */
```

See Also

timer_on, timer_off

The `timer_set` function is not available for the ADSP-21065L chip. Refer to timer0_set, timer1_set to initialize the ADSP-21065L programmable timers.

The function is supplied only as an inlined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_set` must include the `21060.h` header file.

timer0_set, timer1_set

initialize ADSP-21065L processor timers

Synopsis

```
#include <21065l.h>
int timer0_set (unsigned int tperiod,
                unsigned int tcount,
                unsigned int tscale);
int timer1_set (unsigned int tperiod,
                unsigned int tcount,
                unsigned int tscale);
```

Description

The `timer0_set` and `timer1_set` functions set the ADSP-21065L timer registers `TPERIOD0`, `TCOUNT0`, `TPWIDTH0` and `TPERIOD1`, `TCOUNT1`, `TPWIDTH1` respectively. The functions return a 1 if the corresponding timer is enabled, or a zero if the timer is disabled.

Each interrupt call takes approximately 50 cycles on entrance and 50 cycles on return. If `TPERIOD` and `TCOUNT` registers are set too low, you may incur an initializing overhead that could create an infinite loop.

Error Conditions

The `timer0_set` and `timer1_set` functions do not return an error condition.

Example

```
#include <21065l.h>
unsigned int hold_tcount;
if (timer0_set (200, 1, 150) != 1)
    timer0_on ();       /* enable timer 0 */
```

See Also

timer0_off, timer1_off, timer0_on, timer1_on

The functions are supplied only as inlined procedures; that is, the compiler substitutes the appropriate statements for any reference to the procedures. Therefore, any source that references either `timer0_set` or `timer1_set` must include the `21065l.h` header file.

transpm

matrix transpose

Synopsis

```
#include <matrix.h>

float *transpmf (float dm *output,
                 const float dm *a, int rows, int cols);

double *transpm (double dm *output,
                 const double dm *a, int rows, int cols);

long double *transpmd (long double dm *output,
                       const long double dm *a,
                       int rows, int cols);
```

Description

The transpm functions compute the linear algebraic transpose of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, and `output[cols][rows]`.

The algorithm for the linear algebraic transpose of a matrix is defined as:

$$c_{ji} = a_{ij}$$

The functions return a pointer to the output matrix.

Error Conditions

The transpm function do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

float a[ROWS][COLS];
float a_transpose[COLS][ROWS];

transpmf ((float *)(a_transpose),(float *)(a), ROWS, COLS);
```

See Also

No references to this function.

var

variance

Synopsis

```
#include <stats.h>

float varf (const float a[], int n);
double var (const double a[], int n);
long double vard (const long double a[], int n);
```

Description

These functions return the variance of the input array `a[]`. The length of the input array is `n`. The algorithm for computing the variance of a set of data is defined as:

$$c = \frac{n * \sum_{i=0}^{n-1} a_i^{\,2} - (\sum_{i=0}^{n-1} a_i)^2}{n(n-1)}$$

Error Conditions

The var functions do not return an error condition.

Example

```
#include <stats.h>
#define SIZE 256

double data[SIZE];
double result;

result = var (data, SIZE);
```

See Also

mean

vecdot

vector dot product

Synopsis

```
#include <vector.h>

float vecdotf (const float dm a[],
               const float dm b[], int samples);

double vecdot (const double dm a[],
               const double dm b[], int samples);

long double vecdotd (const long double dm a[],
                     const long double dm b[], int samples);
```

Description

The vecdot functions compute the dot product of the vectors `a[]` and `b[]`, which are `samples` in size. They return the scalar result.

The algorithm for calculating the dot product is:

$$return = \sum_{i=0}^{samples\ -1} a_i * b_i$$

where `i = {0,1,2,...,samples-1}`

Error Conditions

The vecdot functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double x[N], y[N];
```

```
double answer;

answer = vecdot (x, y, N);
```

See Also

cvecdot

vecsadd

vector + scalar addition

Synopsis

```
#include <vector.h>

float *vecsaddf (const float dm a[], float scalar,
                 float dm output[], int samples);

double *vecsadd (const double dm a[], double scalar,
                 double dm output[], int samples);

long double *vecsaddd (const long double dm a[],
                       long double scalar,
                       long double dm output[],
                       int samples);
```

Description

The vecsadd functions compute the sum of each element of the vector `a[]`, added to the scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecsadd functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input[N], result[N];
double x;

vecsadd (input, x, result, N);
```

See Also

cvecsadd

vecsmlt

vector * scalar multiplication

Synopsis

```
#include <vector.h>

float *vecsmltf (const float dm a[], float scalar,
                 float dm output[], int samples);

double *vecsmlt (const double dm a[], double scalar,
                 double dm output[], int samples);

long double *vecsmltd (const long double dm a[],
                       long double scalar,
                       long double dm output[],
                       int samples);
```

Description

The vecsmlt functions compute the product of each element of the vector `a[]`, multiplied by the scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecsmlt functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input[N], result[N];
double x;

vecsmlt (input, x, result, N);
```

See Also

cvecsmlt

vecssub

vector - scalar subtraction

Synopsis

```
#include <vector.h>

float *vecssubf (const float dm a[], float scalar,
                 float dm output[], int samples);

double *vecssub (const double dm a[], double scalar,
                 double dm output[], int samples);

long double *vecssubd (const long double dm a[],
                       long double scalar,
                       long double dm output[],
                       int samples);
```

Description

The vecssub functions compute the difference of each element of the vector `a[]`, minus the scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecssub functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input[N], result[N];
double x;

vecssub (input, x, result, N);
```

See Also

cvecssub

vecvadd

vector + vector addition

Synopsis

```
#include <vector.h>

float *vecvaddf (const float dm a[], const float dm b[],
                 float dm output[], int samples);

double *vecvadd (const double dm a[], const double dm b[],
                 double dm output[], int samples);

long double *vecvaddd (const long double dm a[],
                       const long double dm b[],
                       long double dm output[],
                       int samples);
```

Description

The vecvadd functions compute the sum of each of the elements of the vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecvadd functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input_1[N];
double input_2[N], result[N];

vecvadd (input_1, input_2, result, N);
```

See Also

cvecvadd

vecvmlt

vector * vector multiplication

Synopsis

```
#include <vector.h>

float *vecvmltf (const float dm a[], const float dm b[],
                 float dm output[], int samples);

double *vecvmlt (const double dm a[], const double dm b[],
                 double dm output[], int samples);

long double *vecvmltd (const long double dm a[],
                       const long double dm b[],
                       long double dm output[],
                       int samples);
```

Description

The vecvmlt functions compute the product of each of the elements of the vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecvmlt functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input_1[N];
double input_2[N], result[N];

vecvmlt (input_1, input_2, result, N);
```

See Also

cvecvmlt

vecvsub

vector - vector subtraction

Synopsis

```
#include <vector.h>

float *vecvsubf (const float dm a[], const float dm b[],
                 float dm output[], int samples);

double *vecvsub (const double dm a[], const double dm b[],
                 double dm output[], int samples);

long double *vecvsubd (const long double dm a[],
                       const long double dm b[],
                       long double dm output[],
                       int samples);
```

Description

The vecvsub functions compute the difference of each of the elements of the vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecvsub functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input_1[N];
double input_2[N], result[N];

vecvsub (input_1, input_2, result, N);
```

See Also

cvecvsub

zero_cross

count zero crossings

Synopsis

```
#include <stats.h>

int zero_crossf (const float in[], int length);
int zero_cross (const double in[], int length);
int zero_crossd (const long double in[], int length);
```

Description

The zero_cross functions return the number of times that a signal represented in the input array `in[]` crosses over the zero line. If all the input values are either positive or zero, or they are all either negative or zero, then the functions return a zero.

Error Conditions

The zero_cross functions do not return an error condition.

Example

```
#include <stats.h>
#define SIZE 256

double input[SIZE];
int result;

result = zero_cross (input, SIZE);
```

See Also

No references to this function.

5 DSP LIBRARY FOR ADSP-2116X/2126X/2136X PROCESSORS

This chapter describes the DSP run-time library for ADSP-2116x, ADSP-2126x, and ADSP-2136x processors which contains a collection of functions that provide services commonly required by signal processing applications; these functions are in addition to the C/C++ run-time library functions that are described in Chapter 3, "C/C++ Run-Time Library". The services provided by the DSP library functions include support for interrupt handling, signal processing, and access to hardware registers. All these services are Analog Devices extensions to ANSI standard C.

The chapter contains:

- "DSP Run-Time Library Guide" (starting on on page 5-2) contains introductory information about the Analog Devices' special header files and built-in functions that are included with this release of the `cc21k` compiler.
- "DSP Run-Time Library Reference" (starting on on page 5-19) contains the complete reference information for each DSP run-time library function included with this release of the `cc21k` compiler.

For more information on the algorithms on which many of the DSP run-time library's math functions are based, see Cody, W. J. and W. Waite, *Software Manual for the Elementary Function*s, Englewood Cliffs, New Jersey: Prentice Hall, 1980.

DSP Run-Time Library Guide

The DSP run-time library contains routines that you can call from your source program. This section describes how to use the library and provides information on the following topics:

- "Calling DSP Library Functions" on page 5-2
- "Linking DSP Library Functions" on page 5-3
- "Working With Library Source Code" on page 5-4
- "DSP Header Files" on page 5-5
- "Built-In DSP Functions" on page 5-15
- "Implications of Using SIMD Mode" on page 5-17

For information on the contents of the DSP library, see "DSP Run-Time Library Reference" on page 5-19 and on-line Help.

Calling DSP Library Functions

To use a DSP run-time library function, call the function by name and give the appropriate arguments. The names and arguments for each function are described in the function's reference page in the section "DSP Run-Time Library Reference" on page 5-19.

Like other functions you use, library functions should be declared. Declarations are supplied in header files, as described in the section, "Working With Library Source Code" on page 5-4.

C++ namespace prefixing is not supported when calling a DSP library function. All DSP library functions are in the C++ global namespace

The function names are C function names. If you call C run-time library functions from an assembly language program, you must use the assembly version of the function name, which is the function name prefixed with an underscore. For more information on naming conventions, see "C/C++ and Assembly Interface" on page 1-209.

You can use the archiver, described in the *VisualDSP++ 4.0 Linker and Utilities Manual*, to build library archive files of your own functions.

Linking DSP Library Functions

When your C code calls a DSP run-time library function, the call creates a reference that the linker resolves when linking your program. One way to direct the linker to the location of the DSP library is to use the default Linker Description File (`ADSP-21<your_target>.ldf`). The default Linker Description File automatically directs the linker to the appropriate library under your VisualDSP++ installation. Table 5-1 lists the names of these libraries and where they are installed.

Table 5-1. Run-Time Libraries for ADSP-211xx/212xx/213xx Processors

Library Name	Directory	Processor
`libdsp160.dlb`	`211xx\lib`	ADSP-2116x processors, built with `-workaround rframe,21161-anomaly-45`
`libdsp160.dlb`	`211xx\lib\swfa`	ADSP-2116x processors, built with `-workaround rframe,21161-anomaly-45,swfa`
`libdsp26x.dlb`	`2126x\lib`	ADSP-2126x processors
`libdsp26x.dlb`	`2126x\lib\2126x_rev_0.0`	ADSP-2126x processors, built with `-si-revision 0.0`
`libdsp26x.dlb`	`2126x\lib\2126x_rev_any`	ADSP-2126x processors, built with any `-si-revision`
`libdsp36x.dlb`	`213xx\lib`	ADSP-213xx processors

The library located in `212xx\lib` is built without any workarounds enabled; the library in `212xx\lib\212xx_rev_0.0` contains libraries that are suitable for revisions 0.0, 0.1, 0.2; `212xx\lib\212xx_rev_any` contains libraries that will work with all revisions of ADSP-2126x processors.

If an application uses a customized Linker Description File, then either add the appropriate library to the `.LDF` file, or alternatively use the compiler's `-l` switch (`-ldsp160` for ADSP-2116x processors, `-ldsp26x` for ADSP-2126x processors, or `-ldsp36x` for ADSP-2136x processors) to specify that the DSP library is to be added to the link line.

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 `...\21xxx\lib\src`, for ADSP-2116x processors, `...\212xx\lib\src` for ADSP-2126x processors, and `...\213xx\lib\src` for ADSP-2136x processors. The directory contains the source for the C run-time library, for the DSP run-time library, and for the I/O run-time library, as well as the source for the main program start-up functions. If you do not intend to modify any of the run-time library functions, you can delete this directory and its contents to conserve disk space.

The source code is provided so you can customize specific functions for your own needs. To modify these files, you need proficiency in ADSP-21xxx assembly language and an understanding of the run-time environment, as explained in "C/C++ Run-Time Model and Environment" on page 1-175. 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.

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 uses the prototypes to check that each function is called with the correct arguments. The following sections describe the processor-specific header files supplied with this release of the `cc21k` compiler.

21160.h – ADSP-2116x DSP Functions

The `21160.h` header file includes the ADSP-2116x functions of the DSP library, such as `poll_flag_in()`, `timer_set()`, and `idle()`. The `timer_set()`, `timer_on()`, and `timer_off()` functions are also available as inline functions.

21262.h – ADSP-21262 DSP Functions

The `21262.h` header file includes the ADSP-2126x functions of the DSP library, such as `poll_flag_in()`, `timer_set()`, and `idle()`. The `timer_set()`, `timer_on()`, and `timer_off()` functions are also available as inline functions.

21363.h – ADSP-21363 DSP Functions

The `21363.h` header file includes the ADSP-21363 functions of the DSP library, such as `poll_flag_in()`, `timer_set()`, and `idle()`. The `timer_set()`, `timer_on()`, and `timer_off()` functions are also available as inline functions.

21364.h – ADSP-21364 DSP Functions

The `21364.h` header file includes the ADSP-21364 functions of the DSP library, such as `poll_flag_in()`, `timer_set()`, and `idle()`. The `timer_set()`, `timer_on()`, and `timer_off()` functions are also available as inline functions.

21365.h – ADSP-21365 DSP Functions

The `21365.h` header file includes the ADSP-21365 functions of the DSP library, such as `poll_flag_in()`, `timer_set()`, and `idle()`. The `timer_set()`, `timer_on()`, and `timer_off()` functions are also available as inline functions.

asm_sprt.h – Mixed C/Assembly Support

The `asm_sprt.h` header file consists of ADSP-21xxx assembly language macros, not C functions. They are used in your assembly routines that interface with C functions. For more information on this header file, see "Using Mixed C/C++ and Assembly Support Macros" on page 1-214.

cmatrix.h – Complex Matrix Functions

The `cmatrix.h` header file contains prototypes for functions that perform basic arithmetic between two complex matrices, and also between a complex matrix and a complex scalar. The supported complex types are described under the header file `complex.h`.

comm.h – A-law and µ-law Companders

The `comm.h` header file includes the voice-band compression and expansion communication functions as supported by the DSP library for ADSP-2106x processors. However, the functions defined by this header file have not been optimized for the ADSP-2116x/2126x/2136x architectures. Versions of these functions that have been optimized for these architectures are available in the header file `filter.h`. Since these two sets of functions have different arguments, it is important that the user program includes the appropriate header file.

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 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;
```

Additional support for complex numbers is available via the `cmatrix.h` and `cvector.h` header files.

cvector.h – Complex Vector Functions

The `cvector.h` header file contains functions for basic arithmetical operations on vectors of type `complex_float`, `complex_double`, and `complex_long_double`. Support is provided for the dot product operation, as well as for adding, subtracting, and multiplying a vector by either a scalar or vector.

Header Files Defining Processor-Specific System Register Bits

The following header files define symbolic names for processor-specific system register bits. They also contain symbolic definitions for the IOP register address memory and IOP control/status register bits.

def21160.h	ADSP-21160 Bit Definitions
def21161.h	ADSP-21161 Bit Definitions
def21261.h	ADSP-21261 Bit Definitions
def21262.h	ADSP-21262 Bit Definitions
def21266.h	ADSP-21266 Bit Definitions
def21267.h	ADSP-21267 Bit Definitions
def21363.h	ADSP-21363 Bit Definitions
def21364.h	ADSP-21364 Bit Definitions
def21365.h	ADSP-21365 Bit Definitions
def21366.h	ADSP-21366 Bit Definitions
def21367.h	ADSP-21367 Bit Definitions
def21368.h	ADSP-21368 Bit Definitions
def21369.h	ADSP-21369 Bit Definitions

Header Files To Allow Access to Memory Mapped Registers From C/C++ Code

In order to allow safe access to memory-mapped registers from C/C++ code, the header files listed below are supplied. Each memory-mapped register's name is prefixed with "p" and is cast appropriately to ensure the code is generated correctly. For example, SYSCON is defined as follows:

```
#define pSYSCON ((volatile unsigned int *) 0x00)
```

and can be used as:

```
*pSYSCON |= 0x6000;
```

This method of accessing memory-mapped registers should be used in preference to using `asm` statements.

Supplied header files are:

Cdef21160.h	Cdef21161.h	Cdef21261.h	Cdef21262.h
Cdef21266.h	Cdef21267.h	Cdef21363.h	Cdef21364.h
Cdef21365.h	Cdef21366.h	Cdef21367.h	Cdef21368.h
Cdef21369.h			

dma.h – DMA Support Functions

The `dma.h` header file provides definitions and setup, status, enable, and disable functions for DMA operations.

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 Transform (FFT) and convolution. The various forms of the FFT functions provided are `cffN`, `ifftN`, and `rfftN`.

Each form is represented by N separate functions, where `N` represents the number of points supported by the FFT. For example, `cfftN` stands for the functions `cfft65536`, `cfft32768`, and so forth.

The prototypes defined by this header file have been designed to allow the functions to exploit the parallelism within the ADSP-2116x/2126x/2136x architectures. Equivalent functions that have not been optimized for these architectures are supported and are documented in Chapter 4, "DSP Library for ADSP-2106x and ADSP-21020 Processors". These functions are defined in separate header files (`comm.h` for the companding functions, `filters.h` for the filters, and `trans.h` for the FFTs). Since these two sets of functions have different arguments, it is important that the user's program includes the appropriate header file.

The FFT functions are optimized to take advantage of the SIMD (Single Instruction Multiple Data) execution model of the ADSP-2116x, ADSP-2126x, and ADSP-2136x processors, and the function prototypes have been adjusted accordingly.

Complex arguments and complex results must be defined using the `complex_float` data type (defined in the header file `complex.h`). Using the `complex_float` data type ensures that the real and imaginary parts of a complex value are interleaved in memory and therefore are accessible in a single cycle using the wider data bus of the processor.

The FFT functions have also been optimized with respect to memory usage. In general, Fast Fourier Transform algorithms require access to a temporary working buffer. All the FFT functions defined by the `filter.h` header file assume that the input buffer may be corrupted and therefore may be used as a temporary area. This assumption allows the functions to be more efficient both in terms of memory usage and processor speed.

The `cfftN` functions compute the fast Fourier transform of their N-point complex input signal. The `ifftN` functions compute the inverse fast Fourier transform of their N-point complex input signal. The input to each of these functions is a `complex_float` array of N elements. The routines output an N-element array of type `complex_float`. If you wish only to input the real part of a signal, ensure that the imaginary components of the input array are set to zero before calling the function.

The functions first bit-reverse the input arrays and then process them with an optimized block-floating-point FFT (or IFFT) routine.

The `rfftN` functions work like `cfftN` functions, except they operate only on input arrays of real data. This type of operation is equivalent to a `cfftN` function whose imaginary input component is set to zero.

The header file also defines a complex FFT function that has been implemented using a fast radix-2 algorithm. However, this function, `cfftf`, has certain requirements that may not be appropriate for some applications.

filters.h – DSP Filters

The `filters.h` header file includes the signal processing filter functions as supported by the DSP library for ADSP-2106x processors. However, the functions defined by this header file have not been optimized for the ADSP-2116x/2126x/2136x architectures. Versions of these functions that have been optimized for these architectures are available in the `filter.h` header file. Since these two sets of functions have different arguments, it is important that the user program includes the appropriate header file.

macro.h – Circular Buffers

The `macro.h` header file consists of ADSP-21xxx assembly language macros, not C functions. Some are used to manipulate the circular buffer features of the ADSP-21xxx processors.

math.h – Math Functions

The standard math functions defined in the `math.h` header file have been augmented by implementations for the `float` and `long double` data types and some additional functions that are Analog Devices extensions to the ANSI standard.

Table 5-2 provides a summary of the additional library functions defined by the `math.h` header file.

Table 5-2. Math Library - Additional Functions

Description	Prototype

Anti-log	`double alog (double x);`
	`float alogf (float x);`
	`long double alogd (long double x);`

Base 10 Anti-log	`double alog10 (double x);`
	`float alog10f (float x);`
	`long double alog10d (long double x);`

sign copy	`double copysign (double x, double y);`
	`float copysignf (float x, float y);`
	`long double copysignd (long double x, long double y);`

cotangent	`double cot (double x);`
	`float cotf (float x);`
	`long double cotd (long double x);`

average	`double favg (double x, double y);`
	`float favgf (float x, float y);`
	`long double favgd (long double x, long double y);`

clip	`double fclip (double x, double y);`
	`float fclipf (float x, float y);`
	`long double fclipd (long double x, long double y);`

detect Infinity	`int isinf (double x);`
	`int isinff (float x);`
	`int isinfd (long double x);`

detect NaN	`int isnan (double x);`
	`int isnanf (float x);`
	`int isnand (long double x);`

maximum	`double fmax (double x, double y);`
	`float fmaxf (float x, float y);`
	`long double fmaxd (long double x, long double y);`

Table 5-2. Math Library - Additional Functions (Cont'd)

Description	Prototype

minimum	`double fmin (double x, double y);`
	`float fminf (float x, float y);`
	`long double fmind (long double x, long double y);`

reciprocal of square root	`double rsqrt (double x, double y);`
	`float rsqrtf (float x, float y);`
	`long double rsqrtd (long double x, long double y);`

matrix.h – Matrix Functions

The `matrix.h` header file declares a number of function prototypes associated with basic arithmetic operations on matrices of type `float`, `double`, and `long double`. The header file contains support for arithmetic between two matrices, and between a matrix and a scalar.

saturate.h – Saturation Mode Arithmetic

The `saturate.h` header file defines the interface for the saturated arithmetic operations. See "Saturated Arithmetic" on page 1-149 for further information.

sport.h – Serial Port Support Functions

The `sport.h` header file provides definitions and setup, enable, and disable functions for the ADSP-2116x, ADSP-2126x, and ADSP-2136x serial ports.

stats.h – Statistical Functions

The `stats.h` header file includes various statistics functions of the DSP library, such as `mean()` and `autocorr()`.

sysreg.h – Register Access

The `sysreg.h` header file defines a set of built-in functions that provide efficient access to the SHARC system registers from C. The supported functions are fully described in "Access to System Registers" on page 1-103.

trans.h – Fast Fourier Transforms

The `trans.h` header file includes Fast Fourier Transform (FFT) functions as supported by the DSP library for ADSP-2106x processors. However, the functions defined by this header file have not been optimized for the ADSP-2116x/2126x/2136x architectures. Versions of these functions that have been optimized for these architecture are available in the header file `filter.h`. Since these two sets of functions have different arguments, it is important that the user program includes the appropriate header file.

vector.h – Vector Functions

The `vector.h` header file contains functions for operating on vectors of type `float`, `double` and `long double`. Support is provided for the dot product operation and well as for adding, subtracting, and multiplying a vector by either a scalar or vector. Similar support for the complex data types is defined in the header file `cvector.h`.

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 5-3.

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 5-3. Window Generator Functions

Description	Prototype
generate bartlett window	`void gen_bartlett (float w[], int a, int n)`
generate blackman window	`void gen_blackman (float w[], int a, int n)`
generate gaussian window	`void gen_gaussian (float w[], float alpha, int a, int n)`
generate hamming window	`void gen_hamming (float w[], int a, int n)`
generate hanning window	`void gen_hanning (float w[], int a, int n)`
generate harris window	`void gen_harris (float w[], int a, int n)`
generate kaiser window	`void gen_kaiser (float w[], float beta, int a, int n)`
generate rectangular window	`void gen_rectangular (float w[], int a, int n)`
generate triangle window	`void gen_triangle (float w[], int a, int n)`
generate von hann window	`void gen_vonhann (float w[], int a, int n)`

Built-In DSP Functions

The C/C++ compiler supports built-in functions (also known as intrinsic functions) that enable efficient use of hardware resources. Knowledge of these functions is built into the compiler. Your program uses them via normal function call syntax. The compiler notices the invocation and replaces a call to a DSP library function with one or more machine instructions—just as it does for normal operators like “+” and “*”.

Built-in functions are declared in system header files and have names which begin with double underscores, `__builtin`.

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

These functions are specific to individual architectures. The built-in DSP library functions supported at this time on the ADSP-2116x/2126x/2136x architectures are listed in Table 5-4. Refer to "Using Compiler Built-In C Library Functions" on page 3-30 for more information on this topic.

Table 5-4. Built-in DSP Functions

`avg`	`clip`	`copysign`	`copysignf`
`favg`	`favgf`	`fmax`	`fmaxf`
`fmin`	`fminf`	`labs`	`lavg`
`lclip`	`lmax`	`lmin`	`max`
`min`			

Functions `copysign`, `favg`, `fmax`, and `fmin` are compiled as a built-in function only if `double` is the same size as `float`.

Use the `-no-builtin` compiler switch (see on page 1-37) to disable this feature.

The compiler also supports a set of built-in functions for which no inline machine instructions are substituted. This set of built-in functions is characterized by defining one or more pointers in their argument list.

For this set of built-in functions, the compiler relaxes the normal rule whereby any pointer that is passed to a library function must address Data Memory (DM). The compiler recognizes when certain pointers address Program Memory (PM) and generates a call to an appropriate version of the run-time library function. Table 5-5 lists library functions that may be called with pointers that address Program Memory.

Table 5-5. Library Functions Called with Pointers

histogram	matmaddf	matmmltf
matmsubf	matsaddf	matsmltf
matssubf	meanf	rmsf
transpmf	varf	zero_crossf

Use the `-no-builtin` compiler switch (see on page 1-37) to disable this feature.

Implications of Using SIMD Mode

The DSP run-time library for the ADSP-2116x, ADSP-2126x, and ADSP-2136x processors makes extensive use of their SIMD capabilities. In essence, when running in SIMD mode, data contained in memory is always accessed as two 32-bit words, starting at an even word boundary. Therefore, it is essential that any array that is passed to a DSP library function be allocated on a double-word (even word) boundary.

The `cc21k` compiler normally aligns arrays properly in memory. However, the compiler cannot control the allocation of all arrays that are used as arguments to DSP library functions. For example, the alignment of the array `&a[i]` is controlled by the value of the scalar `i`. If the value of the scalar is odd, then the library function will return incorrect results. A variant of this example involves the use of pointers to arrays. If the variable `ptr` is initialized using `ptr=&a[i]` and the value of the scalar `i` is odd, then you cannot use `ptr` to pass an array to a DSP library function.

Refer to "Restrictions to Using SIMD" on page 1-154 for more information on this topic.

A limited number of DSP library functions, whose arguments involve the use of arrays, do not use the SIMD feature of ADSP-2116x, ADSP-2126x, and ADSP-2136x processors due to the nature of their algorithm. These library functions include all `long double` functions, the window generators, as well as:

`biquad`	`cmatmmlt`	`cmatsmlt`
`convolve`	`cvecdot`	`cvecsmlt`
`iir`	`histogram`	`matmmlt`
`matinv`	`transpm`	`zero_cross`

Some ADSP-2116x processor architectures only have a 32-bit external bus and, because of this shorter bus length, the effect of SIMD accesses to external memory on such architectures is not the same as SIMD accesses to internal memory. For this reason, the DSP library contains an alternative set of functions that do not use the architecture's SIMD capabilities. This alternative set is selected in preference to the standard library functions if the `-no-simd` compiler switch (see on page 1-39) is specified at compilation time.

The SIMD feature is described in detail in section "SIMD Support" on page 1-150.

DSP Run-Time Library Reference

The DSP 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. Note the following items that apply to all the functions in the library.

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

Function Benchmarks and Specifications. All functions have been timed from setup, to invocation, to results storage of returned value. This includes all register storing, parameter passing, and so on. Most functions execute slightly faster if you pass constants as arguments instead of variables.

Restrictions. When polymorphic functions are used and the function returns a pointer to program memory, cast the output of the function to pm. For example, `(char pm *)`

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

Name and Purpose of the function

Synopsis – Required header file and functional prototype

Description – Function specification

Error Conditions – Method that the functions use to indicate an error

Example – Typical function usage

See Also – Related functions

a_compress

A-law compression

Synopsis

```
#include <filter.h>
int *a_compress (const int dm input[],
                 int dm output[],
                 int length);
```

Description

The `a_compress` function takes an array of linear 13-bit signed speech samples and compresses them according to ITU recommendation G.711. The array returned contains 8-bit samples that can be sent directly to an A-law codec.

This function returns a pointer to the `output` data array.

The function uses serial port 0 to perform the companding on an ADSP-21160 processor; serial port 0 therefore must not be in use when this routine is called. The serial port is not used by this function on any other ADSP-2116x/2126x/2136x architectures.

Error Conditions

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

Example

```
#include <filter.h>
int data[50], compressed[50];
a_compress (data, compressed, 50);
```

See Also

a_expand, mu_compress

a_expand

A-law expansion

Synopsis

```
#include <filter.h>
int *a_expand (const int dm input[],
               int dm output[],
               int length);
```

Description

The `a_expand` function takes an array of 8-bit compressed speech samples and expands them according to ITU recommendation G.711 (A-law definition). The array returned contains linear 13-bit signed samples. This function returns a pointer to the `output` data array.

The function uses serial port 0 to perform the companding on an ADSP-21160 processor; serial port 0 therefore must not be in use when this routine is called. The serial port is not used by this function on any other ADSP-2116x/2126x/2136x architectures.

Error Conditions

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

Example

```
#include <filter.h>
int expanded_data[50], compressed_data[50];

a_expand (compressed_data, expanded_data, 50);
```

See Also

a_compress, mu_expand

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.

Error Conditions

The input argument `x` for `alogf` must be in the domain [-87.3, 88.7] and the input argument for `alogd` must be in the domain [-708.2, 709.1]. The functions return `HUGE_VAL` if `x` is greater than the domain, and they return 0.0 if `x` is less than the domain.

Example

```
#include <math.h>

double x = 1.0;
double y;
y = alog(x);            /* y = 2.71828... */
```

See Also

alog10, exp, log, pow

alog10

base 10 anti-log

Synopsis

```
#include <math.h>

float alog10f (float x);
double alog10 (double x);
long double alog10d (long double x);
```

Description

The 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))`.

Error Conditions

The input argument `x` for `alog10f` must be in the domain [-37.9 , 38.5] and the input argument for `alog10d` must be in the domain [-307.57, 308.23]. The functions return `HUGE_VAL` if `x` is greater than the domain, and they return 0.0 if `x` is less than the domain.

Example

```
#include <math.h>

double x = 1.0;
double y;
y = alog10(x);          /* 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. The phase of a Cartesian number is computed as:

$$c = atan\left(\frac{\mathrm{Im}(a)}{\mathrm{Re}(a)}\right)$$

Error Conditions

The arg functions return a zero if `a.re` <> 0 and `a.im` = 0.

Example

```
#include <complex.h>

complex_float x = {0.0,1.0};
float r;
r = argf(x);       /* r = π/2 */
```

See Also

atan2, cartesian, polar

autocoh

autocoherence

Synopsis

```
#include <stats.h>

float *autocohf (float dm out[],
                 const float dm in[],
                 int samples, int lags);

double *autocoh (double dm out[],
                 const double dm in[],
                 int samples, int lags);

long double *autocohd (long double dm out[],
                       const long double dm in[],
                       int samples, int lags);
```

Description

The autocoh functions compute the autocoherence of the floating-point input, `in[]`, which contain `samples` values. The autocoherence of an input signal is its autocorrelation minus its mean squared. The functions return a pointer to the output array `out[]` of length `lags`.

Error Conditions

The autocoh functions do not return an error condition.

Example

```
#include <stats.h>
#define SAMPLES 1024
#define LAGS      16

double excitation[SAMPLES];
double response[LAGS];
```

```
int lags = LAGS;

autocoh (response, excitation, SAMPLES, lags);
```

See Also

autocorr, crosscoh, crosscorr

By default, the `autocohf` function (and `autocoh`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

autocorr

autocorrelation

Synopsis

```
#include <stats.h>

float *autocorrf (float dm out[], const float dm in[],
                  int samples, int lags);

double *autocorr (double dm out[], const double dm in[],
                  int samples, int lags);

long double *autocorrd (long double dm out[],
                        const long double dm in[],
                        int samples, int lags);
```

Description

The autocorr functions perform an autocorrelation of a signal. Autocorrelation is the cross-correlation of a signal with a copy of itself. It provides information about the time variation of the signal. The signal to be autocorrelated is given by the `in[]` input array. The number of samples of the autocorrelation sequence to be produced is given by `lags`. The length of the input sequence is given by `samples`. The functions return a pointer to the `out[]` output data array of length `lags`.

Autocorrelation is used in digital signal processing applications such as speech analysis.

Error Conditions

The autocorr functions do not return an error condition.

Example

```
#include <stats.h>
#define SAMPLES 1024
```

```
#define LAGS      16

double excitation[SAMPLES];
double response[LAGS];
int lags = LAGS;

autocorr (response, excitation, SAMPLES, lags);
```

See Also

autocoh, crosscoh, crosscorr

By default, the `autocorrf` function (and `autocorr`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

biquad

biquad filter section

Synopsis

```
#include <filter.h>

float *biquad (const float dm input[],
               float dm output[],
               const float pm coeffs[],
               float dm state[],
               int samples,
               int sections);
```

Description

The `biquad` function implements a cascaded biquad filter defined by the coefficients and delay line that are supplied in the call of `biquad`. The function generates the filtered response of the input data `input` and stores the result in the output vector `output`. The number of input samples and the length of the output vector is specified by the argument `samples`.

The characteristics of the filter are dependent upon the filter coefficients and the number of biquad sections. The number of sections is specified by the argument `sections`, and the filter coefficients are supplied to the function using the argument `coeffs`. Each stage has five coefficients that must be stored in the order `A2` , `A1` , `B2` , `B1` , `B0` . The value of `A0` is assumed to be `1.0`, and `A1` and `A2` should be scaled accordingly.

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]`.

The `coeffs` array must be allocated in program memory (PM) as the function uses the single-cycle dual-memory fetch of the processor. The definition of the `coeffs` array is therefore:

```
float pm coeffs[5*sections];
```

Each filter should have its own delay line which is represented by the array `state`. The `state` array should be large enough for two delay elements per biquad section and to hold an internal pointer that allows the filter to be restarted. The definition of the state is:

```
float dm state[2*sections + 1];
```

The `state` array should be initially cleared to zero before calling the function for the first time and should not otherwise be modified by the user program.

The function returns a pointer to the output vector.

The algorithm used is adapted from *Digital Signal Processing*, Oppenheim and Schafer, New Jersey, Prentice Hall, 1975.

Error Conditions

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

Example

```
#include <filter.h>
#define NSECTIONS  4
#define NSAMPLES  64
#define NSTATE    (2*NSECTIONS) + 1

float input[NSAMPLES];
float output[NSAMPLES];
float state[NSTATE];
float pm coeffs[5*NSECTIONS];
int i;
```

```
for (i = 0; i < NSTATE; i++)
    state[i] = 0;      /* initialize state array */

biquad (input, output, coeffs, state, NSAMPLES, NSECTIONS);
```

See Also

fir, iir

cabs

complex absolute value

Synopsis

```
#include <complex.h>
float cabsf (complex_float z);
double cabs (complex_double z);
long double cabsd (complex_long_double z);
```

Description

The cabs functions return the floating-point absolute value of their complex input.

The absolute value of a complex number is evaluated with the following formula.

$$y = \sqrt{((Re(z))^2 + (Im(z))^2)}$$

Error Conditions

The cabs functions do not return an error condition.

Example

```
#include <complex.h>
complex_float cnum;
float answer;

cnum.re = 12.0;
cnum.im = 5.0;

answer = cabsf (cnum);      /* answer = 13.0 */
```

See Also

fabs, labs

cadd

complex addition

Synopsis

```
#include <complex.h>
complex_float caddf (complex_float a, complex_float b);
complex_double cadd (complex_double a, complex_double b);
complex_long_double caddd (complex_long_double a,
                           complex_long_double b);
```

Description

The cadd functions add the two complex values `a` and `b` together, and return the result.

Error Conditions

The cadd functions do not return any error conditions.

Example

```
#include <complex.h>

complex_double x = {9.0,16.0};
complex_double y = {1.0,-1.0};
complex_double z;

z = cadd (x,y);     /* z.re = 10.0, z.im = 15.0 */
```

See Also

cmlt, csub

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 it returns as the function's result, and a phase that is returned via the second argument `phase`.

The formula for converting from Cartesian to polar notation is given by:

```
magnitude = cabs(a)
phase = arg(a)
```

Error Conditions

The cartesian functions return a zero for the phase if `a.re` <> 0 and `a.im` = 0.

Example

```
#include <complex.h>

complex_float point = {-2.0 , 0.0};
float phase;
```

```
float mag;
mag = cartesianf (point,&phase);      /* mag = 2.0, phase = π */
```

See Also

arg, cabs, polar

cdiv

complex division

Synopsis

```
#include <complex.h>

complex_float cdivf (complex_float a, complex_float b);
complex_double cdiv (complex_double a, complex_double b);
complex_long_double cdivd (complex_long_double a,
                           complex_long_double b);
```

Description

The cdiv functions compute the complex division of complex input `a` by complex input `b`, and return the result.

Error Conditions

The cdiv functions set both the real and imaginary parts of the result to Infinity if `b` is equal to `(0.0,0.0)` .

Example

```
#include <complex.h>

complex_double x = {3.0,11.0};
complex_double y = {1.0, 2.0};
complex_double z;

z = cdiv (x,y);     /* z.re = 5.0, z.im = 1.0 */
```

See Also

cadd, cmlt, csub

cexp

complex exponential

Synopsis

```
#include <complex.h>
complex_float cexpf (complex_float z);
complex_double cexp (complex_double z);
complex_long_double cexpd (complex_long_double z);
```

Description

The cexp functions compute the complex exponential value e to the power of the first argument. The exponential of a complex value is evaluated with the following formula.

```
Re(y) = exp (Re(z)) * cos (Im(z));
Im(y) = exp (Re(z)) * sin (Im(z));
```

Error Conditions

For underflow errors, the cexp functions return zero.

Example

```
#include <complex.h>

complex_float cnum;
complex_float answer;

cnum.re = 1.0;
cnum.im = 0.0;

answer = cexpf (cnum);      /* answer = (2.7182 + 0i) */
```

See Also

log, pow

cfft_mag

cfft magnitude

Synopsis

```
#include <filter.h>

float *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 `cfftN` function. The size of the signal and the size of the power spectrum is `fftsize` .

The algorithm used to calculate the normalized power spectrum is:

$$\text{magnitude}(z) = \frac{\sqrt{\text{Re}(z)^2 + \text{Im}(z)^2}}{\text{fftsize}}$$

The function returns a pointer to the `output` matrix.

The Nyquist frequency is located at `(fftsize/2) + 1`.

Error Conditions

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

Example

```
#include <filter.h>
#define N 64

complex_float fft_input[N];
complex_float fft_output[N];
float spectrum[N];
```

```
cfft64 (fft_input, fft_output);
cfft_mag (fft_output, spectrum, N);
```

See Also

cfftN, rfft_mag

By default, this function uses SIMD.
Refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

cfftN

N-point complex input FFT

Synopsis

```
#include <filter.h>

complex_float *cfft65536 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft32768 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft16384 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft8192  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft4096  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft2048  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft1024  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft512   (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft256   (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft128   (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft64    (complex_float dm input[],
                          complex_float dm output[]);
```

```
complex_float *cfft16    (complex_float dm input[],
                          complex_float dm output[]);

complex_float *cfft8     (complex_float dm input[],
                          complex_float dm output[]);
```

Description

The `cfftN` functions are optimized to take advantage of the SIMD execution model of the ADSP-2116x/2126x/2136x processors, and require complex arguments to ensure that the real and imaginary parts are interleaved in memory and are therefore accessible in a single cycle using the wider data bus of the processor.

Each of these 14 `cfftN` functions computes the N-point radix-2 fast Fourier transform (CFFT) of its complex input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are fourteen distinct functions in this set. They all perform the same function with the same type and number of arguments. The only difference between them is the size of the arrays they operate on. Call a particular function by substituting the number of points for N, as in

```
cfft8 (input, output);
```

The input to `cfftN` is a floating-point array of N points. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. Better results occur with less zero padding, however. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data. Optimum memory usage can be achieved by specifying the input array as the output array, but at the cost of some run-time performance.

The `cfftN()` function returns a pointer to the `output` array.

Error Conditions

The `cfftN` functions do not return any error conditions.

Example

```
#include <filter.h>
#define N 2048

complex_float input[N], output[N];

   /* Input array is filled from a converter or other source */

cfft2048 (input, output);
   /* Array is filled with FFT data */
```

See Also

cfft_mag, cfftf, ifftN, rfftN

The `cfftN` functions use the input array as an intermediate workspace. If the input data is to be preserved it must first be copied to a safe location before calling these functions.

By default, these functions use SIMD.
Refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

cfftf

fast N-point complex input FFT

Synopsis

```
#include <filter.h>

void cfftf (float data_real[], float data_imag[],
            float temp_real[]. float temp_imag[],
            const float twid_real[],
            const float twid_imag[],
            int n);
```

Description

The `cfftf` 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 "decimates in frequency" using an optimized radix-2 algorithm.

The array `data_real` contains the real part of a complex input signal, and the array `data_imag` contains the imaginary part of the signal. On output, the function overwrites the data in these arrays and stores the real part of the FFT in `data_real`, and the imaginary part of the FFT in `data_imag`. If the input data is to be preserved, it must first be copied to a safe location before calling this function. The argument `n` represents the number of points in the FFT; it must be a power of 2 and must be at least 64.

The `cfftf` function has been designed for optimum performance and requires that the arrays `data_real` and `data_imag` are aligned on an address boundary that is a multiple of the FFT size. For certain applications, this alignment constraint may not be appropriate; in such cases, the application should call the `cfftN` function instead with no loss of facility (apart from performance).

The arrays `temp_real` and `temp_imag` are used as intermediate temporary buffers and should each be of size `n`.

The twiddle table is passed in using the arrays `twid_real` and `twid_imag` . The array `twid_real` contains the `+cosine` factors, and the array `twid_imag` contains the `-sine` factors; each array should be of size `n/2` . The `twidfftf` function may be used to initialize the twiddle table arrays.

It is recommended that the arrays containing real parts (`data_real`, `temp_real`, and `twid_real`) are allocated in separate memory blocks from the arrays containing imaginary parts (`data_imag`, `temp_imag`, and `twid_imag`); otherwise, the performance of the function degrades.

The `cfftf` function has been highly optimized and should therefore not be used with any application that relies on the `-reserve` switch (see on page 1-47) for correct operation.

Error Conditions

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

Example

```
#include <filter.h>

#define FFT_SIZE 1024

#pragma align 1024
float dm input_r[FFT_SIZE];
#pragma align 1024
float pm input_i[FFT_SIZE];

float dm temp_r[FFT_SIZE];
float pm temp_i[FFT_SIZE];
float dm twid_r[FFT_SIZE/2];
float pm twid_i[FFT_SIZE/2];

twidfftf(twid_r,twid_i,FFT_SIZE);
```

```
cfftf(input_r,input_i,
      temp_r,temp_i,
      twid_r,twid_i,FFT_SIZE);
```

See Also

cfftN, ifftN, rfftN, twidfftf

The `cfftf` function has been implemented to make highly efficient use of the ADSP-2116x processor's SIMD capabilities. The DSP library therefore does not contain a version of this function that does not use SIMD.
Refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

cmatmadd

complex matrix + matrix addition

Synopsis

```
#include <cmatrix.h>

complex_float *cmatmaddf (complex_float dm *output,
                          const complex_float dm *a,
                          const complex_float dm *b,
                          int rows, int cols);

complex_double *cmatmadd (complex_double dm *output,
                          const complex_double dm *a,
                          const complex_double dm *b,
                          int rows, int cols);

complex_long_double *cmatmaddd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          const complex_long_double dm *b,
                          int rows, int cols);
```

Description

The cmatmadd functions perform a complex matrix addition of the input-matrix `a[][]` with input complex matrix `b[][]`, and store the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The cmatmadd functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (double_complex *) (&a);
complex_double b[ROWS][COLS], *b_p = (double_complex *) (&b);
complex_double c[ROWS][COLS], *res_p = (double_complex *) (&c);

cmatmadd (res_p, a_p, b_p, ROWS, COLS);
```

See Also

cmatmsub, cmatsadd

By default, the `cmatmaddf` function (and `cmatmadd`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

cmatmmlt

complex matrix * matrix multiplication

Synopsis

```
#include <cmatrix.h>

complex_float *cmatmmltf (complex_float dm *output,
                          const complex_float dm *a,
                          const complex_float dm *b,
                          int a_rows, int a_cols, int b_cols);

complex_double *cmatmmlt (complex_double dm *output,
                          const complex_double dm *a,
                          const complex_double dm *b,
                          int a_rows, int a_cols, int b_cols);

complex_long_double *cmatmmltd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          const complex_long_double dm *b,
                          int a_rows, int a_cols, int b_cols);
```

Description

The cmatmmlt functions perform a complex matrix multiplication of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[a_rows][a_cols]`, `b[a_cols][b_cols]`, and `output[a_rows][b_cols]`. The functions return a pointer to the output matrix.

Complex matrix multiplication is defined by the following algorithm:

$$\mathrm{Re}(c_{i,j}) = \sum_{l=0}^{a_cols-1} (\mathrm{Re}(a_{i,l}) * \mathrm{Re}(b_{l,j}) - \mathrm{Im}(a_{i,l}) * \mathrm{Im}(b_{l,j}))$$

$$\mathrm{Im}(c_{i,j}) = \sum_{l=0}^{a_cols-1} (\mathrm{Re}(a_{i,l}) * \mathrm{Im}(b_{l,j}) + \mathrm{Im}(a_{i,l}) * \mathrm{Re}(b_{l,j}))$$

where i={0,1,2,...,a_rows-1}, j={0,1,2,...,b_cols-1}

Error Conditions

The cmatmmlt functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS_1 4
#define COLS_1 8
#define COLS_2 2

complex_double a[ROWS_1][COLS_1], *a_p = (double_complex *) (&a);
complex_double b[COLS_1][COLS_2], *b_p = (double_complex *) (&b);
complex_double c[ROWS_1][COLS_2], *r_p = (double_complex *) (&c);

cmatmadd (r_p, a_p, b_p, ROWS_1, COLS_1, COLS_2);
```

See Also

cmatsmlt

cmatmsub

complex matrix - matrix subtraction

Synopsis

```
#include <cmatrix.h>

complex_float *cmatmsubf (complex_float dm *output,
                          const complex_float dm *a,
                          const complex_float dm *b,
                          int rows, int cols);

complex_double *cmatmsub (complex_double dm *output,
                          const complex_double dm *a,
                          const complex_double dm *b,
                          int rows, int cols);

complex_long_double *cmatmsubd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          const complex_long_double dm *b,
                          int rows, int cols);
```

Description

The cmatmsub functions perform a complex matrix subtraction between the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The cmatmsub functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (double_complex *) (&a);
complex_double b[ROWS][COLS], *b_p = (double_complex *) (&b);
complex_double c[ROWS][COLS], *res_p = (double_complex *) (&c);

cmatmsub (res_p, a_p, b_p, ROWS, COLS);
```

See Also

cmatmadd, cmatssub

By default, the `cmatmsubf` function (and `cmatmsub`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

cmatsadd

complex matrix + scalar addition

Synopsis

```
#include <cmatrix.h>

complex_float *cmatsaddf (complex_float dm *output,
                          const complex_float dm *a,
                          complex_float scalar,
                          int rows, int cols);

complex_double *cmatsadd (complex_double dm *output,
                          const complex_double dm *a,
                          complex_double scalar,
                          int rows, int cols);

complex_long_double *cmatsaddd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          complex_long_double scalar,
                          int rows, int cols);
```

Description

The cmatsadd functions add a complex scalar to each element of the complex input matrix `a[][]` and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The cmatsadd functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
```

```
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (double_complex *) (&a);
complex_double c[ROWS][COLS], *res_p = (double_complex *) (&c);
complex_double z;

cmatsadd (res_p, a_p, z, ROWS, COLS);
```

See Also

cmatmadd, cmatssub

By default, the `cmatsaddf` function (and `cmatsadd`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

cmatsmlt

complex matrix * scalar multiplication

Synopsis

```
#include <cmatrix.h>

complex_float *cmatsmltf (complex_float dm *output,
                          const complex_float dm *a,
                          complex_float scalar
                          int rows, int cols);

complex_double *cmatsmlt (complex_double dm *output,
                          const complex_double dm *a,
                          complex_double scalar,
                          int rows, int cols);

complex_long_double *cmatsmltd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          complex_long_double scalar,
                          int rows, int cols);
```

Description

The cmatsmlt functions multiply each element of the complex input matrix `a[][]` with a complex scalar, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Complex matrix by scalar multiplication is defined by the following algorithm:

$$\mathrm{Re}(c_{i,j}) = \mathrm{Re}(a_{i,j}) * \mathrm{Re}(scalar) - \mathrm{Im}(a_{i,j}) * \mathrm{Im}(scalar)$$
$$\mathrm{Im}(c_{i,j}) = \mathrm{Re}(a_{i,j}) * \mathrm{Im}(scalar) + \mathrm{Im}(a_{i,j}) * \mathrm{Re}(scalar)$$

where i={0,1,2,..., rows-1}, j={0,1,2,..., cols-1}

Error Conditions

The cmatsmlt functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (double_complex *) (&a);
complex_double c[ROWS][COLS], *res_p = (double_complex *) (&c);
complex_double z;

cmatsmlt (res_p, a_p, z, ROWS, COLS);
```

See Also

cmatmmlt

cmatssub

complex matrix - scalar subtraction

Synopsis

```
#include <cmatrix.h>

complex_float *cmatssubf (complex_float dm *output,
                          const complex_float dm *a,
                          complex_float scalar,
                          int rows, int cols);

complex_double *cmatssub (complex_double dm *output,
                          const complex_double dm *a,
                          complex_double scalar,
                          int rows, int cols);

complex_long_double *cmatssubd (complex_long_double dm *output,
                          const complex_long_double dm *a,
                          complex_long_double scalar,
                          int rows, int cols);
```

Description

The cmatssub functions subtract a complex scalar from each element of the complex input matrix `a[][]` and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The cmatssub functions do not return an error condition.

Example

```
#include <cmatrix.h>
#define ROWS 4
```

```
#define COLS 8

complex_double a[ROWS][COLS], *a_p = (double_complex *) (&a);
complex_double c[ROWS][COLS], *res_p = (double_complex *) (&c);
complex_double z;

cmatssub (res_p, a_p, z, ROWS, COLS);
```

See Also

cmatmsub, cmatsadd

By default, the `cmatssubf` function (and `cmatssub`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

cmlt

complex multiplication

Synopsis

```
#include <complex.h>

complex_float cmltf (complex_float a, complex_float b);
complex_double cmlt (complex_double a, complex_double b);
complex_long_double cmltd (complex_long_double a,
                           complex_long_double b);
```

Description

The cmlt functions compute the complex multiplication of the complex numbers `a` and `b`, and return the result.

Error Conditions

The cmlt functions do not return any error conditions.

Example

```
#include <complex.h>

complex_float x = {3.0,11.0};
complex_float y = {1.0, 2.0};
complex_float z;

z = cmltf(x,y);     /* z.re = 14.0, z.im = 22.0 */
```

See Also

cadd, cdiv, csub

conj

complex conjugate

Synopsis

```
#include <complex.h>

complex_float conjf (complex_float a);
complex_double conj (complex_double a);
complex_long_double conjd (complex_long_double a);
```

Description

These functions conjugate the complex input `a`, and return the result.

Error Conditions

The conj functions do not return any error conditions.

Example

```
#include <complex.h>

complex_double x = {2.0,8.0};
complex_double z;

z = conj(x);     /* z = (2.0,-8.0) */
```

See Also

No references to this function.

convolve

convolution

Synopsis

```
#include <filter.h>
float *convolve (const float a[], int asize,
                 const float b[], int bsize, float output);
```

Description

This function calculates the convolution of the input vectors `a[]` and `b[]`, and returns the result in the vector `output[]`. The lengths of these vectors are `a[asize]`, `b[bsize]`, and `output[asize+bsize-1]`.

The function returns a pointer to the output vector.

Convolution of two vectors is defined as:

$$c_k = \sum_{j=m}^{n} a_j * b_{(k-j)}$$

where

```
k = {0, 1, ..., asize + bsize - 2}
m = max( 0, k + 1 - bsize)
n = min( k, asize - 1)
```

Error Conditions

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

Example

```
#include <filter.h>
```

```
float input[81];
float response[31];
float output[81 + 31 -1];

convolve(input,81,response,31,output);
```

See Also

crosscoh, ifftN, rfftN

copysign

copy the sign of the floating-point operand (IEEE arithmetic function)

Synopsis

```
#include <math.h>
double copysign (double x, double y);
float copysignf (float x, float y);
long double copysignd (long double x, long double y);
```

Description

The copysign functions copy the sign of the second argument `y` to the first argument `x` without changing either its exponent or mantissa.

The `copysignf` function is a built-in function which is implemented with an `Fn=Fx COPYSIGN Fy` instruction. The `copysign` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

These functions do not return an error code.

Example

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

x = copysign (0.5, -10.0);        /* x = -0.5 */
y = copysignf (-10.0, 0.5f);      /* y = 10.0 */
```

See Also

No references to this function.

cot

cotangent

Synopsis

```
#include <math.h>
double cot (double x);
float cotf (float x);
long double cotd (long double x);
```

Description

The cot functions return the cotangent of their argument. The input is interpreted as radians.

Error Conditions

The input argument `x` for `cotf` must be in the domain [-1.647e6, 1.647e6] and the input argument for `cotd` must be in the domain [-4.21657e8, 4.21657e8]. The functions return zero if `x` is outside their domain.

Example

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

y = cot (x);
v = cotf (w);
```

See Also

tan

crosscoh

cross-coherence

Synopsis

```
#include <stats.h>

float *crosscohf (float dm out[],
                  const float dm x[], const float dm y[],
                  int samples, int lags);

double *crosscoh (double dm out[],
                  const double dm x[], const double dm y[],
                  int samples, int lags);

long double *crosscohd (long double dm out[],
                        const long double dm x[],
                        const long double dm y[],
                        int samples, int lags);
```

Description

The crosscoh functions compute the cross-coherence of two floating-point inputs, `x[]` and `y[]`. The cross-coherence is the cross-correlation minus the product of the mean of `x` and the mean of `y`. The length of the input arrays is given by `samples`. The functions return a pointer to the output array `out[]` of length `lags`.

Error Conditions

The crosscoh functions do not return an error condition.

Example

```
#include <stats.h>
#define SAMPLES 1024
#define LAGS      16
```

```
double excitation[SAMPLES], y[SAMPLES];
double response[LAGS];
int lags = LAGS;

crosscoh (response, excitation, y, SAMPLES, lags);
```

See Also

autocoh, autocorr, crosscorr

By default, the `crosscohf` function (and `crosscoh`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

crosscorr

cross-correlation

Synopsis

```
#include <stats.h>

float *crosscorrf (float dm out[],
                   const float dm x[], const float dm y[],
                   int samples, int lags);

double *crosscorr (double dm out[],
                   const double dm x[], const double dm y[],
                   int samples, int lags);

long double *crosscorrd (long double dm out[],
                         const long double dm x[],
                         const long double dm y[],
                         int samples, int lags);
```

Description

The crosscorr functions perform a cross-correlation between two signals. The cross-correlation is the sum of the scalar products of the signals in which the signals are displaced in time with respect to one another. The signals to be correlated are given by the input arrays `x[]` and `y[]`. The length of the input arrays is given by `samples`. The functions return a pointer to the output data array `out[]` of length `lags`.

Cross-correlation is used in signal processing applications such as speech analysis.

Error Conditions

The crosscorr functions do not return an error condition.

Example

```
#include <stats.h>
#define SAMPLES 1024
#define LAGS      16

double excitation[SAMPLES], y[SAMPLES];
double response[LAGS];
int lags = LAGS;

crosscorr (response, excitation, y, SAMPLES, lags);
```

See Also

autocoh, autocorr, crosscoh

By default, the `crosscorrf` function (and `crosscorr`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

csub

complex subtraction

Synopsis

```
#include <complex.h>
complex_float csubf(complex_float a, complex_float b);
complex_double csub (complex_double a, complex_double b);
complex_long_double csubd (complex_long_double a,
                           complex_long_double b);
```

Description

The csub functions subtract the two complex values `a` and `b`, and return the result.

Error Conditions

The csub functions do not return any error conditions.

Example

```
#include <complex.h>

complex_float x = {9.0,16.0};
complex_float y = {1.0,-1.0};
complex_float z;

z = csubf(x,y);     /* z.re = 8.0, z.im = 17.0 */
```

See Also

cadd, cdiv, cmlt

cvecdot

complex vector dot product

Synopsis

```
#include <cvector.h>

complex_float cvecdotf (const complex_float dm a[],
                        const complex_float dm b[], int samples);

complex_double cvecdot (const complex_double dm a[],
                        const complex_double dm b[], int samples);

complex_long_double cvecdotd (const complex_long_double dm a[],
                              const complex_long_double dm b[],
                              int samples);
```

Description

The cvecdot functions compute the complex dot product of the complex vectors `a[]` and `b[]`, which are `samples` in size. The scalar result is returned by the function.

The algorithm for a complex dot product is given by:

$$\mathrm{Re}(c_i) = \sum_{l=0}^{n-1} \mathrm{Re}(a_i) * \mathrm{Re}(b_i) - \mathrm{Im}(a_i) * \mathrm{Im}(b_i)$$

$$\mathrm{Im}(c_i) = \sum_{l=0}^{n-1} \mathrm{Re}(a_i) * \mathrm{Im}(b_i) + \mathrm{Im}(a_i) * \mathrm{Re}(b_i)$$

Error Conditions

The cvecdot functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float x[N], y[N];
complex_float answer;

answer = cvecdotf (x, y, N);
```

See Also

vecdot

cvecsadd

complex vector + scalar addition

Synopsis

```
#include <cvector.h>

complex_float *cvecsaddf (const complex_float dm a[],
                          complex_float scalar,
                          complex_float dm output[], int samples);

complex_double *cvecsadd (const complex_double dm a[],
                          complex_double scalar,
                          complex_double dm output[], int samples);

complex_long_double *cvecsaddd (const complex_long_double dm a[],
                                complex_long_double scalar,
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecsadd functions compute the sum of each element of the complex vector `a[]`, added to the complex scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The cvecsadd functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input[N], result[N];
complex_float x;
```

```
cvecsaddf (input, x, result, N);
```

See Also

vecvadd

By default, the `cvecsaddf` function (and `cvecsadd`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

cvecsmlt

complex vector * scalar multiply

Synopsis

```
#include <cvector.h>

complex_float *cvecsmltf (const complex_float dm a[],
                          complex_float scalar,
                          complex_float dm output[], int samples);

complex_double *cvecsmlt (const complex_double dm a[],
                          complex_double scalar,
                          complex_double dm output[], int samples);

complex_long_double *cvecsmltd (const complex_long_double dm a[],
                                complex_long_double scalar,
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecsmlt functions compute the product of each element of the complex vector `a[]`, multiplied by the complex scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Complex vector by scalar multiplication is given by the formula:

$$\mathrm{Re}(c_i) = \mathrm{Re}(a_i)*\mathrm{Re}(scalar) - \mathrm{Im}(a_i)*\mathrm{Im}(scalar)$$

$$\mathrm{Im}(c_i) = \mathrm{Re}(a_i)*\mathrm{Im}(scalar) + \mathrm{Im}(a_i)*\mathrm{Re}(scalar)$$

where: $i=\{0,1,2,\ldots,samples-1\}$

Error Conditions

The cvecsmlt functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input[N], result[N];
complex_float x;

cvecsmltf (input, x, result, N);
```

See Also

vecvmlt

cvecssub

complex vector - scalar subtraction

Synopsis

```
#include <cvector.h>

complex_float *cvecssubf (const complex_float dm a[],
                          complex_float scalar,
                          complex_float dm output[], int samples);

complex_double *cvecssub (const complex_double dm a[],
                          complex_double scalar,
                          complex_double dm output[], int samples);

complex_long_double *cvecssubd (const complex_long_double dm a[],
                                complex_long_double scalar,
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecssub functions compute the difference of each element of the complex vector `a[]`, minus the complex scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The cvecssub functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input[N], result[N];
complex_float x;
```

```
cvecssubf (input, x, result, N);
```

See Also

vecvsub

By default, the `cvecssubf` function (and `cvecssub`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

cvecvadd

complex vector + vector addition

Synopsis

```
#include <cvector.h>

complex_float *cvecvaddf (const complex_float dm a[],
                          const complex_float dm b[],
                          complex_float dm output[], int samples);

complex_double *cvecvadd (const complex_double dm a[],
                          const complex_double dm b[],
                          complex_double dm output[], int samples);

complex_long_double *cvecvaddd (const complex_long_double dm a[],
                                const complex_long_double dm b[],
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecvadd functions compute the sum of each of the elements of the complex vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The cvecvadd functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input_1[N];
complex_float input_2[N], result[N];
```

```
cvecvaddf (input_1, input_2, result, N);
```

See Also

vecvadd

By default, the `cvecvaddf` function (and `cvecvadd`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

cvecvmlt

complex vector * vector multiply

Synopsis

```
#include <cvector.h>

complex_float *cvecvmltf (const complex_float dm a[],
                          const complex_float dm b[],
                          complex_float dm output[], int samples);

complex_double *cvecvmlt (const complex_double dm a[],
                          const complex_double dm b[],
                          complex_double dm output[], int samples);

complex_long_double *cvecvmltd (const complex_long_double dm a[],
                                const complex_long_double dm b[],
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecvmlt functions compute the product of each of the elements of the complex vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size.The functions return a pointer to the output vector.

Complex vector multiplication is given by the formula:

$$\mathrm{Re}(c_i) = \mathrm{Re}(a_i)*\mathrm{Re}(b_i) - \mathrm{Im}(a_i)*\mathrm{Im}(b_i)$$
$$\mathrm{Im}(c_i) = \mathrm{Re}(a_i)*\mathrm{Im}(b_i) + \mathrm{Im}(a_i)*\mathrm{Re}(b_i)$$

where: `i={0,1,2,...,samples-1}`

Error Conditions

The cvecvmlt functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input_1[N];
complex_float input_2[N], result[N];

cvecvmltf (input_1, input_2, result, N);
```

See Also

vecvmlt

By default, the `cvecvmltf` function (and `cvecvmlt`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

cvecvsub

complex vector - vector subtraction

Synopsis

```
#include <cvector.h>

complex_float *cvecvsubf (const complex_float dm a[],
                          const complex_float dm b[],
                          complex_float dm output[], int samples);

complex_double *cvecvsub (const complex_double dm a[],
                          const complex_double dm b[],
                          complex_double dm output[], int samples);

complex_long_double *cvecvsubd (const complex_long_double dm a[],
                                const complex_long_double dm b[],
                                complex_long_double dm output[],
                                int samples);
```

Description

The cvecvsub functions compute the difference of each of the elements of the complex vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The cvecvsub functions do not return an error condition.

Example

```
#include <cvector.h>
#define N 100

complex_float input_1[N];
complex_float input_2[N], result[N];
```

```
cvecvsubf (input_1, input_2, result, N);
```

See Also

vecvsub

By default, the `cvecvsubf` function (and `cvecvsub`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

favg

mean of two values

Synopsis

```
#include <math.h>

double favg (double x, double y);
float favgf (float x, float y);
long double favgd (long double x, long double y);
```

Description

The favg functions return the mean of their two arguments.

The `favgf` function is a built-in function which is implemented with an `Fn=(Fx+Fy)/2` instruction. The `favg` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The favg functions do not return an error code.

Example

```
#include <math.h>

float x;
x = favgf (10.0f, 8.0f);     /* returns 9.0f */
```

See Also

avg, lavg

fclip

clip

Synopsis

```
#include <math.h>

double fclip (double x, double y);
float fclipf (float x, float y);
long double fclipd (long double x, long double y);
```

Description

The fclip functions return the 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 an `Fn=CLIP Fx BY Fy` instruction. The `fclip` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The fclip functions do not return an error code.

Example

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

y = fclipf (5.1f, 8.0f);     /* returns 5.1f */
```

See Also

clip, lclip

fir

finite impulse response (FIR) filter

Synopsis

```
#include <filter.h>
float *fir (const float dm input[],
            float dm output[],
            const float pm coeffs[],
            float dm state[],
            int samples,
            int taps);
```

Description

The `fir` function is optimized to take advantage of the SIMD execution model of the ADSP-2116x/2126x/2136x processors.

The `fir` function implements a finite impulse response (FIR) filter defined by the coefficients and delay line that are supplied in the call of `fir`. The function produces the filtered response of its input data and stores the result in the vector `output`. This FIR filter is structured as a sum of products. The characteristics of the filter (passband, stop band, etc.) are dependent on the coefficient values and the number of taps supplied by the calling program.

The floating-point input array to the filter is `samples` in length. The integer `taps` indicates the length of the filter, which is also the length of the array `coeffs`. The `coeffs` array holds one FIR filter coefficient per element. The coefficients are stored in reverse order, and so `coeffs[0]` contains the last coefficient and `coeffs[taps-1]` contains the first coefficient. The array must be located in program memory data space so that the single-cycle dual-memory fetch of the processor can be used.

The `state` array contains a pointer to the delay line as its first element, followed by the delay line values. The length of the `state` array is therefore 1 greater than the number of `taps`.

Each filter has its own `state` array, which should not be modified by the calling program, only by the `fir` function. The state array should be initialized to zeros before the `fir` function is called for the first time. The function returns a pointer to the output vector.

Error Conditions

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

Example

```
#include <filter.h>

#define TAPS 10

float x[100], y[100];
float pm coeffs[TAPS];      /* coeffs array must be          */
                            /* initialized and in PM memory */
float state[TAPS+1];
int i;

for (i = 0; i < TAPS+1; i++)
    state[i] = 0;           /* initialize state array       */

fir (x, y, coeffs, state, 100, TAPS);
                            /* y holds the filtered output  */
```

See Also

biquad, iir

By default, this function uses SIMD.
Refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

fmax

maximum

Synopsis

```
#include <math.h>

double fmax (double x, double y);
float fmaxf (float x, float y);
long double fmaxd (long double x, long double y);
```

Description

The fmax functions return the larger of their two arguments.

The `fmaxf` function is a built-in function which is implemented with an `Fn=MAX(Fx,Fy)` instruction. The `fmax` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The fmax functions do not return an error code.

Example

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

y = fmaxf (5.1f, 8.0f);      /* returns 8.0f */
```

See Also

fmin, lmax, lmin, max, min

fmin

minimum

Synopsis

```
#include <math.h>

double fmin (double x, double y);
float fminf (float x, float y);
long double fmind (long double x, long double y);
```

Description

The fmin functions return the smaller of their two arguments.

The `fminf` function is a built-in function which is implemented with an `Fn=MIN(Fx,Fy)` instruction. The `fmin` function is compiled as a built-in function if `double` is the same size as `float`.

Error Conditions

The fmin functions do not return an error code.

Example

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

y = fminf (5.1f, 8.0f);      /* returns 5.1f */
```

See Also

fmax, lmax, lmin, max, min

gen_bartlett

generate bartlett window

Synopsis

```
#include <window.h>
void gen_bartlett(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                         */);
```

Description

The `gen_bartlett` function generates a vector containing the Bartlett window. The length is specified by parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

The Bartlett window is similar to the Triangle window (see "gen_triangle" on page 5-99) but has the following different properties

- The Bartlett window always returns a window with two zeros on either end of the sequence. Therefore, for odd `n`, the center section of a `N+2` Bartlett window equals a `N` Triangle window.
- For even `n`, the Bartlett window is the convolution of two rectangular sequences. There is no standard definition for the Triangle window for even `n`; the slopes of the Triangle window are slightly steeper than those of the Bartlett window.

Algorithm

$$w[n] = 1 - \left| \frac{n - \frac{N-1}{2}}{\frac{N-1}{2}} \right|$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_blackman

generate blackman window

Synopsis

```
#include <window.h>
void gen_blackman(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_blackman` 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, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_gaussian

generate gaussian window

Synopsis

```
#include <window.h>
void gen_gaussian(
float dm w[], /* Window vector                                    */
float alpha,  /* Gaussian alpha parameter                         */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_gaussian` 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, ..., N-1} and α is an input parameter

Domain

$a > 0$; $N > 0$; $\alpha > 0.0$

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_hamming

generate hamming window

Synopsis

```
#include <window.h>
void gen_hamming(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_hamming` function generates a vector containing the Hamming window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Algorithm

$$w[n] = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right)$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_hanning

generate hanning window

Synopsis

```
#include <window.h>
void gen_hanning(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_hanning` function generates a vector containing the Hanning window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`. This window is also known as the Cosine window.

Algorithm

$$w[n] = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right)$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_harris, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_harris

generate harris window

Synopsis

```
#include <window.h>
void gen_harris(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                         */);
```

Description

The `gen_harris` 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, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_kaiser, gen_rectangular, gen_triangle, gen_vonhann

gen_kaiser

generate kaiser window

Synopsis

```
#include <window.h>
void gen_kaiser(
float dm w[], /* Window vector                                    */
float beta,   /* Kaiser beta parameter                           */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_kaiser` 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, ..., N-1}, α= (N - 1) /2, and $I_0(\beta)$ represents the zeroth-order modified Bessel function of the first kind.

Domain

a > 0; N > 0; β > 0.0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_rectangular, gen_triangle, gen_vonhann

gen_rectangular

generate rectangular window

Synopsis

```
#include <window.h>
void gen_rectangular(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_rectangular` function generates a vector containing the Rectangular window. The length of the window required is specified by the parameter `N`, and the stride parameter `a` is used to space the window values within the output vector `w`. The length of the output vector should therefore be `N*a`.

Algorithm

w[n] = 1

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_triangle, gen_vonhann

gen_triangle

generate triangle window

Synopsis

```
#include <window.h>
void gen_triangle(
float dm w[], /* Window vector                                    */
int a,        /* Address stride in samples for window vector */
int N         /* Length of window vector                        */);
```

Description

The `gen_triangle` 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*a`.

Refer to the Bartlett window (described on page 5-89) regarding the relationship between it and the Triangle window.

Algorithm

For even `n`, the following equation applies:

$$w[n] = \begin{cases} \dfrac{2n+1}{N} & n < N/2 \\ \dfrac{2N-2n-1}{N} & n > N/2 \end{cases}$$

where n = {0, 1, 2, ..., N-1}

For odd `n`, the following equation applies:

$$w[n] = \begin{cases} \dfrac{2n+2}{N+1} & n < N/2 \\ \dfrac{2N-2n}{N+1} & n > N/2 \end{cases}$$

where n = {0, 1, 2, ..., N-1}

Domain

a > 0; N > 0

Error Conditions

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

See Also

gen_bartlett, gen_blackman, gen_gaussian, gen_hamming, gen_hanning, gen_harris, gen_kaiser, gen_rectangular, gen_vonhann

gen_vonhann

generate von hann window

Synopsis

```
#include <window.h>
void gen_vonhann(
float dm 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 5-94).

Error Conditions

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

See Also

gen_hanning

histogram

histogram

Synopsis

```
#include <stats.h>
int *histogram (int out[],
                const int in[],
                int out_len,
                int samples,
                int bin_size);
```

Description

The `histogram` function computes a scaled-integer histogram of its input array. The `bin_size` parameter is used to adjust the width of each individual bin in the output array. For example, a `bin_size` of 5 indicates that the first location of the output array holds the number of occurrences of a `0`, `1`, `2`, `3` or `4`.

The output array is first zeroed by the function, then each sample in the input array is multiplied by `1/bin_size` and truncated. The appropriate bin in the output array is incremented. This function returns a pointer to the output array.

For maximum performance, this function does not perform out of bounds checking. Therefore, all values within the input array must be within range (that is, between 0 and `bin_size * out_len`).

Error Conditions

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

Example

```
#include <stats.h>
#define SAMPLES 1024
```

```
int length = 2048;
int excitation[SAMPLES], response[2048];
histogram (response, excitation, length, SAMPLES, 5);
```

See Also

mean, var

idle

execute ADSP-21xxx processor `idle` instruction

Synopsis

```
#include <21160.h>
#include <21262.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>
void idle (void);
```

Description

The `idle` function invokes the processor's `idle` instruction once and returns. The `idle` instruction causes the processor to stop and respond only to interrupts. For a complete description of the `idle` instruction, please refer to the appropriate SHARC Processor Hardware Reference manual.

In earlier releases of the VisualDSP++ software (prior to release 2.1), the `idle` function repeatedly executed the `idle` instruction. This function has been changed to give you more control over the amount of time spent in the `idle` state.

Error Conditions

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

Example

```
#include <21160.h>
idle ();
```

See Also

interrupt, signal

ifftN

N-point inverse FFT

Synopsis

```
#include <filter.h>
complex_float *ifft65536 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft32768 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft16384 (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft8192  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft4096  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft2048  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft1024  (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft512   (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft256   (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft128   (complex_float input[],
                          complex_float dm output[]);

complex_float *ifft64    (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft32    (complex_float dm input[],
                          complex_float dm output[]);
```

```
complex_float *ifft16    (complex_float dm input[],
                          complex_float dm output[]);

complex_float *ifft8     (complex_float dm input[],
                          complex_float dm output[]);
```

Description

The `ifftN` functions are optimized to take advantage of the SIMD execution model of the ADSP-2116x/2126x/2136x processors. They require complex arguments to ensure that the real and imaginary parts are interleaved in memory and are therefore accessible in a single cycle using the wider data bus of the processor.

Each of these fourteen `ifftN` functions computes the N-point radix-2 inverse fast Fourier transform (IFFT) of its floating-point input (where N is 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are 14 distinct functions in this set. They all perform the same function with the same type and number of arguments. The only difference between them is the size of the arrays on which they operate. To call a particular function, substitute the number of points for N. For example,

```
ifft8 (input, output);
```

The input to `ifftN` is a floating-point array of N points. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. However, better results occur with less zero padding. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data. Optimum memory usage can be achieved by specifying the input array as the output array, but at the cost of some run-time performance.

The `ifftN` functions return a pointer to the `output` array.

Error Conditions

The `ifftN` functions do not return error conditions.

Example

```
#include <filter.h>
#define N 2048

complex_float input[N], output[N];

   /* Input array is filled from a previous xfft2048 () or
   other source */

ifft2048 (input, output);
   /* Arrays are filled with FFT data */
```

See Also

cfftN, rfftN

The `ifftN` functions use the input array as an intermediate workspace. If the input data is to be preserved it must first be copied to a safe location before calling these functions.

By default, these functions use SIMD.
Refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

iir

infinite impulse response (IIR) filter

Synopsis

```
#include <filter.h>
float *iir (const float dm input[],
            float dm output[],
            const float pm coeffs[],
            float dm state[],
            int samples,
            int sections);
```

Description

The `iir` function implements an infinite impulse response (IIR) filter defined by the coefficients and delay line that are supplied in the call of `iir`. The filter is implemented as a cascaded biquad, and generates the filtered response of the input data `input` and stores the result in the output vector `output`. The number of input samples and the length of the output vector is specified by the argument `samples`.

The characteristics of the filter are dependent upon the filter coefficients and the number of biquad sections. The number of sections is specified by the argument `sections`, and the filter coefficients are supplied to the function using the argument `coeffs`. Each stage has four coefficients which must be ordered in the following form:

```
[a2 stage 1, a1 stage 1, b2 stage 1, b1 stage 1, a2 stage 2, ...]
```

The function assumes that the value of `B0` is `1.0`, and so the `B1` and `B2` coefficients should be scaled accordingly. As a consequence of this, all the output generated by the `iir` function has to be scaled by the product of all the `B0` coefficients to obtain the correct signal amplitude. The function also assumes that the value of the `A0` coefficient is `1.0`, and the `A1` and `A2` coefficients should be normalized. These requirements are demonstrated in the **Example** below.

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]`.

The `coeffs` array must be allocated in program memory (PM) as the function uses the single-cycle dual-memory fetch of the processor. The definition of the `coeffs` array is therefore:

```
float pm coeffs[4*sections];
```

Each filter should have its own delay line which is represented by the array `state`. The `state` array should be large enough for two delay elements per biquad section and to hold an internal pointer that allows the filter to be restarted. The definition of the state is:

```
float state[2*sections + 1];
```

The `state` array should be initially cleared to zero before calling the function for the first time and should not otherwise be modified by the user program.

The function returns a pointer to the output vector.

The algorithm used is adapted from *Digital Signal Processing*, Oppenheim and Schafer, New Jersey, Prentice Hall, 1975.

Algorithm

$$H(z) = \prod_{n=0}^{\text{sections}-1} \frac{1 + (b_n1/b_n0)z^{-1} + (b_n2/b_n0)z^{-2}}{1 + (a_n1/a_n0)z^{-1} + (a_n2/a_n0)z^{-2}}$$

To get the correct amplitude of the signal, H(z) should be adjusted by:

$$H(z) = H(z) * \prod_{n=0}^{\text{sections}-1} \frac{b_n0}{a_n0}$$

Error Conditions

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

Example

```
#include <filter.h>

#define SAMPLES 100
#define SECTIONS 4

/* Coefficients generated by a filter design tool that uses
   a transposed direct form II */

const struct {
   float a0;
   float a1;
   float a2;
} A_coeffs[SECTIONS];

const struct {
   float b0;
   float b1;
   float b2;
} B_coeffs[SECTIONS];

/* Coefficients for the iir function */

float pm coeffs[4 * SECTIONS];

/* Input, Output, and State Arrays */

float input[SAMPLES], output[SAMPLES];
float state[2*SECTIONS + 1];

float scale;      /* used to scale the output from iir */

/* Utility Variables */
float a0,a1,a2;
float b0,b1,b2;
```

```
int i;

/* Transform the A-coefficients and B-coefficients from a filter
   design tool into coefficients for the iir function */

scale = 1.0;

for (i = 0; i < SECTIONS; i++) {

   a0 = A_coeffs[i].a0;
   a1 = A_coeffs[i].a1;
   a2 = A_coeffs[i].a2;

   coeffs[(i*4) + 0] = -(a2/a0);
   coeffs[(i*4) + 1] = -(a1/a0);

   b0 = B_coeffs[i].b0;
   b1 = B_coeffs[i].b1;
   b2 = B_coeffs[i].b2;

   coeffs[(i*4) + 2] = (b2/b0);
   coeffs[(i*4) + 3] = (b1/b0);

   scale = scale * (b0/a0);
}

/* Call the iir function */

for (i = 0; i <= 2*SECTIONS; i++)
   state[i] = 0;          /* initialize the state array */

iir (input, output, coeffs, state, SAMPLES, SECTIONS);

/* Adjust output by all (b0/a0) terms */

for (i = 0; i < SAMPLES; i++)
   output[i] = output[i] * scale;
```

See Also

biquad, fir

matinv

real matrix inversion

Synopsis

```
#include <matrix.h>

float *matinvf (float dm *output,
                const float dm *input, int samples);

double *matinv (double dm *output,
                const double dm *input, int samples);

long double *matinvd (long double dm *output,
                      const long double dm *input, int samples);
```

Description

The matinv functions employ Gauss-Jordan elimination with full pivoting to compute the inverse of the input matrix `input` and store the result in the matrix `output`. The dimensions of the matrices `input` and `output` are `[n][n]`. The functions return a pointer to the output matrix.

Error Conditions

If no inverse exists for the input matrix, the functions return a null pointer.

Example

```
#include <matrix.h>
#define N 8

double a[N][N};
double a_inv[N][N};

matinv ((double *)(a_inv),(double *)(a),N);
```

See Also

No references available.

matmadd

real matrix + matrix addition

Synopsis

```
#include <matrix.h>

float *matmaddf (float dm *output,
                 const float dm *a,
                 const float dm *b, int rows, int cols);

double *matmadd (double dm *output,
                 const double dm *a,
                 const double dm *b, int rows, int cols);

long double *matmaddd (long double dm *output,
                const long double dm *a,
                const long double dm *b, int rows, int cols);

float *matadd  (float dm *output,
                const float dm *a,
                const float dm *b, int rows, int cols);
```

Description

The matmadd functions perform a matrix addition of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`.

The functions return a pointer to the output matrix.

The `matadd` function is equivalent to `matmaddf` and is provided for compatibility with previous versions of VisualDSP++.

Error Conditions

The matmadd functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input_1[ROWS][COLS], *a_p = (double *) (&input_1);
double input_2[ROWS][COLS], *b_p = (double *) (&input_2);
double result[ROWS][COLS], *res_p = (double *) (&result);

matmadd (res_p, a_p, b_p, ROWS, COLS);
```

See Also

matmsub

By default, the `matmaddf` function (and `matmadd`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

matmmlt

real matrix * matrix multiplication

Synopsis

```
#include <matrix.h>

float *matmmltf (float dm *output,
                 const float dm *a,
                 const float dm *b,
                 int a_rows, int a_cols, b_cols);

double *matmmlt (double dm *output,
                 const double dm *a,
                 const double dm *b,
                 int a_rows, int a_cols, b_cols);

long double *matmmltd (long double dm *output,
                 const long double dm *a,
                 const long double dm *b,
                 int a_rows, int a_cols, b_cols);

float *matmul (float dm *output,
                 const float dm *a,
                 const float dm *b,
                 int a_rows, int a_cols, b_cols);
```

Description

The matmmlt functions perform a matrix multiplication of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[a_rows][a_cols]`, `b[a_cols][b_cols]`, and `output[a_rows][b_cols]`.

The functions return a pointer to the output matrix.

Matrix multiplication is defined by the following algorithm:

$$c_{i,j} = \sum_{l=0}^{a_cols-1} a_{i,l} * b_{l,j}$$

where `i={0,1,2,...,a_rows-1}, j={0,1,2,...,b_cols-1}`

The `matmul` function is equivalent to `matmmltf` and is provided for compatibility with previous versions of VisualDSP++.

Error Conditions

The matmmlt functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS_1 4
#define COLS_1 8
#define COLS_2 2

double input_1[ROWS_1][COLS_1], *a_p = (double *) (&input_1);
double input_2[COLS_1][COLS_2], *b_p = (double *) (&input_2);
double result[ROWS_1][COLS_2], *res_p = (double *) (&result);

matmmlt (res_p, a_p, b_p, ROWS_1, COLS_1, COLS_2);
```

See Also

matsmlt

matmsub

real matrix - matrix subtraction

Synopsis

```
#include <matrix.h>

float *matmsubf (float dm *output,
                 const float dm *a,
                 const float dm *b, int rows, int cols);

double *matmsub (double dm *output,
                 const double dm *a,
                 const double dm *b, int rows, int cols);

long double *matmsubd (long double dm *output,
                 const long double dm *a,
                 const long double dm *b, int rows, int cols);

float *matsub  (float dm *output,
                 const float dm *a,
                 const float dm *b, int rows, int cols);
```

Description

The matmsub functions perform a matrix subtraction of the input matrices `a[][]` and `b[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, `b[rows][cols]`, and `output[rows][cols]`.

The functions return a pointer to the output matrix.

The `matsub` function is equivalent to `matmsubf` and is provided for compatibility with previous versions of VisualDSP++.

Error Conditions

The matmsub functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input_1[ROWS][COLS], *a_p = (double *) (&input_1);
double input_2[ROWS][COLS], *b_p = (double *) (&input_2);
double result[ROWS][COLS], *res_p = (double *) (&result);

matmsub (res_p, a_p, b_p, ROWS, COLS);
```

See Also

matmadd

By default, the `matmsubf` function (and `matmsub`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

matsadd

real matrix + scalar addition

Synopsis

```
#include <matrix.h>

float *matsaddf (float dm *output, const float dm *a,
                 float scalar, int rows, int cols);

double *matsadd (double dm *output, const double dm *a
                 double scalar, int rows, int cols);

long double *matsaddd (long double dm *output,
                       const long double dm *a,
                       long double scalar, int rows, int cols);
```

Description

The matsadd functions add a scalar to each element of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The matsadd functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input[ROWS][COLS], *a_p = (double *) (&input);
double result[ROWS][COLS], *res_p = (double *) (&result);
double x;
```

```
matsadd (res_p, a_p, x, ROWS, COLS);
```

See Also

matmadd, matssub

By default, the `matsaddf` function (and `matsadd`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

matsmlt

real matrix * scalar multiplication

Synopsis

```
#include <matrix.h>

float *matsmltf (float dm *output, const float dm *a,
                 float scalar, int rows, int cols);

double *matsmlt (double dm *output, const double dm *a
                 double scalar, int rows, int cols);

long double *matsmltd (long double dm *output,
                       const long double dm *a,
                       long double scalar, int rows, int cols);

float *matscalmult (float dm *output, const float dm *a,
                    float scalar, int rows, int cols);
```

Description

The matsmlt functions multiply a scalar with each element of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`.

The functions return a pointer to the output matrix.

The `matscalmult` function is equivalent to `matsmltf` and is provided for compatibility with previous versions of VisualDSP++.

Error Conditions

The matsmlt functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input[ROWS][COLS], *a_p = (double *) (&input);
double result[ROWS][COLS], *res_p = (double *) (&result);
double x;

matsmlt (res_p, a_p, x, ROWS, COLS);
```

See Also

matmmlt

By default, the `matsmltf` function (and `matsmlt`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

matssub

real matrix - scalar subtraction

Synopsis

```
#include <matrix.h>

float *matssubf (float dm *output, const float dm *a,
                 float scalar, int rows, int cols);

double *matssub (double dm *output, const double dm *a
                 double scalar, int rows, int cols);

long double *matssubd (long_double dm *output,
                       const long double dm *a,
                       long double scalar, int rows, int cols);
```

Description

The matssub functions subtract a scalar from each element of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]` and `output[rows][cols]`. The functions return a pointer to the output matrix.

Error Conditions

The matssub functions do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

double input[ROWS][COLS], *a_p = (double *) (&input);
double result[ROWS][COLS], *res_p = (double *) (&result);
double x;
```

```
matssub (res_p, a_p, x, ROWS, COLS);
```

See Also

matmsub, matsadd

By default, the `matssubf` function (and `matssub`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

mean

mean

Synopsis

```
#include <stats.h>

float meanf (const float in[], int length);
double mean (const double in[], int length);
long double meand (const long double in[], int length);
```

Description

These functions return the mean of the input array `in[]`. The length of the input array is `length`.

Error Conditions

The mean functions do not return an error condition.

Example

```
#include <stats.h>
#define SIZE 256

double data[SIZE];
double result;

result = mean (data, SIZE);
```

See Also

var

By default, the `meanf` function (and `mean`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

mu_compress

µ-law compression

Synopsis

```
#include <filter.h>
int *mu_compress (const int dm input[],
                  int dm output[]
                  int samples);
```

Description

The `mu_compress` function takes an array of linear 14-bit signed speech samples and compresses them according to ITU recommendation G.711. The output array returned contains 8-bit samples that can be sent directly to a µ-law codec.

The function returns a pointer to the compressed data.

The function uses serial port 0 to perform the companding on an ADSP-21160 processor; serial port 0 therefore must not be in use when this routine is called. The serial port is not used by this function on any other ADSP-2116x, ADSP-2126x or ADSP-2136x architectures.

Error Conditions

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

Example

```
#include <filter.h>
int linear[100], compressed[100];

mu_compress (linear, compressed, 100);
```

See Also

a_compress, mu_expand

mu_expand

µ-law expansion

Synopsis

```
#include <filter.h>
int *mu_expand (const int dm input[],
                int dm output[],
                int samples);
```

Description

The `mu_expand` function takes an array of 8-bit compressed speech samples and expands them according to ITU recommendation G.711 (µ-law definition). The output returned is an array of linear 14-bit signed samples. The function returns a pointer to the output array.

The function uses serial port 0 to perform the companding on an ADSP-21160 processor; serial port 0 therefore must not be in use when this routine is called. The serial port is not used by this function on any other ADSP-2116x, ADSP-2126x or ADSP-2136x architectures.

Error Conditions

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

Example

```
#include <filter.h>
int compressed_data[100], expanded_data[100];

mu_expand (compressed_data, expanded_data, 100);
```

See Also

a_expand, mu_compress

norm

normalization

Synopsis

```
#include <complex.h>

complex_float normf (complex_float a);
complex_double norm (complex_double a);
complex_long_double normfd(complex_long_double a);
```

Description

These functions normalize the complex input `a` and return the result. Normalization of a complex number is defined as:

$$\mathrm{Re}(c) = \frac{\mathrm{Re}(a)}{\sqrt{\mathrm{Re}^2(a) + \mathrm{Im}^2(a)}}$$

$$\mathrm{Im}(c) = \frac{\mathrm{Im}(a)}{\sqrt{\mathrm{Re}^2(a) + \mathrm{Im}^2(a)}}$$

Error Conditions

The norm functions return zero if `cabs(a)` is equal to zero.

Example

```
#include <complex.h>

complex_double x = {2.0,-4.0};
complex_double z;

z = norm(x);      /* z = (0.4472,-0.8944) */
```

See Also

No references to this function.

polar

construct from polar coordinates

Synopsis

```
#include <complex.h>

complex_float polarf (complex_float mag, complex_float phase);
complex_double polar (complex_double mag, complex_double phase);
complex_long_double polard (complex_long_double mag,
                            complex_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.

The algorithm for transforming a polar coordinate into a Cartesian coordinate is:

$\mathrm{Re}(c) = mag * \cos(phase)$
$\mathrm{Im}(c) = mag * \sin(phase)$

Error Conditions

The polar functions do not return any error conditions.

Example

```
#include <complex.h>
#define PI 3.14159265

float magnitude = 2.0;
float phase = PI;
```

```
complex_float z;

z = polarf (magnitude,phase);      /* z.re = -2.0, z.im = 0.0 */
```

See Also

arg, cartesian

poll_flag_in

test input flag

Synopsis

```
#include <21160.h>
#include <21262.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>
int poll_flag_in (int flag, int mode);
```

Description

The `poll_flag_in` function tests the specified flag (0, 1, 2, 3) for the specified transition (0=low to high, 1=high to low, 2=flag high, 3=flag low, 4=any transition, 5=read flag). The function returns a zero *after* the specified transition has occurred in modes 0-3. In Mode 4, it returns the state of the flag after the transition. In Mode 5, it returns the value of the flag without waiting.

Table 5-6. poll_flag_in Macros and Values

Flag Macro	Value	Mode Macro	Value
READ_FLAG0	0	FLAG_IN_LO_TO_HI	0
READ_FLAG1	1	FLAG_IN_HI_TO_LOW	1
READ_FLAG2	2	FLAG_IN_HI	2
READ_FLAG3	3	FLAG_IN_LOW	3
READ_FLAG3	3	FLAG_IN_TRANSITION	4
READ_FLAG3	3	RETURN_FLAG_STATE	5

This function assumes that the flag direction in the `MODE2` register is already set as an input (the default state at reset).

Error Conditions

The `poll_flag_in` function returns a negative value for an invalid flag or transition mode.

Example

```
#include <21160.h>
poll_flag_in (0, 3);
             /* return zero after transition has occurred */
```

See Also

interrupt, set_flag

rfft_mag

rfft magnitude

Synopsis

```
#include <filter.h>

float *rfft_mag (const complex_float dm input[],
                 float dm output[],
                 int fftsize);

float *fft_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. The size of the signal and the size of the power spectrum is `fftsize/2`.

The algorithm used to calculate the normalized power spectrum is:

$$\text{magnitude}(z) = \frac{2\sqrt{\text{Re}(z)^2 + \text{Im}(z)^2}}{\text{fftsize}}$$

The function returns a pointer to the `output` matrix.

The `fft_mag` function is equivalent to `rfft_mag` and is provided for compatibility with previous versions of VisualDSP++.

The Nyquist frequency is located at `fftsize/2`.

Error Conditions

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

Example

```
#include <filter.h>
#define N 64

float fft_input[N];
complex_float fft_output[N/2];
float spectrum[N/2];

rfft64 (fft_input, fft_output);
rfft_mag (fft_output, spectrum, N);
```

See Also

cfft_mag, rfftN

By default, this function uses SIMD.
Refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

rfftN

N-point real input FFT

Synopsis

```
#include <filter.h>
complex_float *rfft65536 (float dm input[],
                          complex_float dm output[]);

complex_float *rfft32768 (float dm input[],
                          complex_float dm output[]);

complex_float *rfft16384 (float dm input[],
                          complex_float dm output[]);

complex_float *rfft8192  (float dm input[],
                          complex_float dm output[]);

complex_float *rfft4096  (float dm input[],
                          complex_float dm output[]);

complex_float *rfft2048  (float dm input[],
                          complex_float dm output[]);

complex_float *rfft1024  (float dm input[],
                          complex_float dm output[]);

complex_float *rfft256   (float dm input[],
                          complex_float dm output[]);

complex_float *rfft128   (float dm input[],
                          complex_float dm output[]);

complex_float *rfft64    (float dm input[],
                          complex_float dm output[]);

complex_float *rfft32    (float dm input[],
                          complex_float dm output[]);

complex_float *rfft16    (float dm input[],
                          complex_float dm output[]);
```

Description

The `rfftN` functions are optimized to take advantage of the SIMD execution model of the ADSP-2116x/2126x/2136x processors, and require complex output results to ensure that the real and imaginary parts are interleaved in memory and are therefore accessible in a single cycle using the wider data bus of the processor.

Each of these `rfftN` functions are similar to the `cfftN` functions except that they only take real inputs. They compute the N-point radix-2 fast Fourier transform (RFFT) of their floating-point input (where N is 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536).

There are thirteen distinct functions in this set. They all perform the same function with the same type and number of arguments. The only difference between them is the size of the arrays on which they operate.

Call a particular function by substituting the number of points for N, as in the following example.

```
rfft16 (input, output);
```

The input to `rfftN` is a floating-point array of N points. If there are fewer than N actual data points, you must pad the array with zeros to make N samples. However, better results occur with less zero padding. The input data should be windowed (if necessary) before calling the function because no preprocessing is performed on the data.

The complex frequency domain signal generated by the `rfftN` functions is stored in the array output. Because the output signal is symmetric around the midpoint of the frequency domain, the functions only generate `N/2` output points.

The rfftN functions do not calculate the Nyquist frequency (which would normally located at output `[N/2]`). The cfftN functions should be used in place of these functions if the Nyquist frequency is required.

The `rfftN` functions return a pointer to the output array.

Error Conditions

The `rfftN` functions do not return any error conditions.

Example

```
#include <filter.h>
#define N 2048

float input[N];
complex_float output[N/2];

   /* Real input array fills from a converter or other source */

rfft2048 (input, output);
   /* Arrays are filled with FFT data */
```

See Also

cfftN, cfftf, ifftN, rfft_mag

The `rfftN` functions use the input array as an intermediate workspace. If the input data is to be preserved it must first be copied to a safe location before calling these functions.

By default, these functions use SIMD.
Refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

rms

root mean square

Synopsis

```
#include <stats.h>

float rmsf (const float in[], int length);
double rms (const double in[], int length);
long double rmsd (const long double in[], int length);
```

Description

These functions return the square root of the mean of the square of the input array `in[]`. The length of the input array is `length`.

Error Conditions

The rms functions do not return an error condition.

Example

```
#include <stats.h>
#define SIZE 256

double data[SIZE];
double result;

result = rms (data, SIZE);
```

See Also

mean, var

By default, the `rmsf` function (and `rms`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

rsqrt

reciprocal square root

Synopsis

```
#include <math.h>
double rsqrt (double x);
float rsqrtf (float x);
long double rsqrtd (long double x);
```

Description

The rsqrt functions return the reciprocal positive square root of their argument.

Error Conditions

The rsqrt functions return zero for a negative input.

Example

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

y = rsqrt (2.0);      /* y = 0.707 */
```

See Also

sqrt

set_flag

set ADSP-21xxx processor flags

Synopsis

```
#include <21160.h>
#include <21262.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>
int set_flag (int flag, int mode);
```

Description

The `set_flag` function is used to set the ADSP-21xxx processor flags to the desired output value.

The function accepts as input a flag number [0-3] and a mode. The mode can be specified as a macro (defined in `21160.h`) or a value [0-3].

Table 5-7. Flag Function Macros and Values

Flag Macro	Value	Mode Macro	Value
SET_FLAG0	0	SET_FLAG	0
SET_FLAG1	1	CLR_FLAG	1
SET_FLAG2	2	TGL_FLAG	2
SET_FLAG3	3	TST_FLAG	3

In addition to setting the flag to the specified value, the function also sets the `MODE2` register to specify that the flag is used for output, not input.

If the `TST_FLAG` macro (or a 3) is specified as the mode, the current value (0 or 1) of the flag is returned as the result of the function.

The `set_flag` function returns a zero upon success (except as noted in the previous paragraph).

Error Conditions

The `set_flag` function returns a non-zero for an error.

Example

```
#include <21160.h>

set_flag (SET_FLAG0, CLR_FLAG);
set_flag (SET_FLAG0, SET_FLAG);
```

See Also

poll_flag_in

set_semaphore

set bus lock semaphore

Synopsis

```
#include <21160.h>
#include <21262.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>
int set_semaphore (
    void dm *semaphore, int set_value, int timeout);
```

Description

The `set_semaphore` function is used to control bus lock in multiprocessor ADSP-21xxx systems.

- -1 is returned if the bus is locked and the bus lock timeout is exceeded.
- 0 (zero) is returned if the bus is not locked and a semaphore is set.

Error Conditions

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

See Also

No references to this function.

timer_off

disable ADSP-21xxx processor timer

Synopsis

```
#include <21160.h>
#include <21262.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>
unsigned int timer_off (void);
```

Description

The `timer_off` function disables the ADSP-21xxx timer and returns the current value of the `TCOUNT` register.

Error Conditions

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

Example

```
#include <21160.h>
unsigned int hold_tcount;

hold_tcount = timer_off ();
              /* hold_tcount contains value of TCOUNT */
              /* register AFTER timer has stopped     */
```

See Also

timer_on, timer_set

The function is supplied only as an inlined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_off` must include an appropriate header file.

timer_on

enable ADSP-21xxx processor timer

Synopsis

```
#include <21160.h>
#include <21262.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>
unsigned int timer_on (void);
```

Description

The `timer_on` function enables the ADSP-21xxx timer and returns the current value of the `TCOUNT` register.

Error Conditions

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

Example

```
#include <21160.h>
unsigned int hold_tcount;

hold_tcount = timer_on ();
              /* hold_tcount contains value of TCOUNT */
              /* register when timer starts           */
```

See Also

timer_off, timer_set

The function is supplied only as an inlined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_on` must include an appropriate header file.

timer_set

initialize ADSP-21xxx processor timer

Synopsis

```
#include <21160.h>
#include <21262.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>
int timer_set (unsigned int tperiod,
               unsigned int tcount);
```

Description

The `timer_set` function sets the ADSP-21xxx timer registers `TPERIOD` and `TCOUNT`. The function returns a 1 if the timer is enabled, or a zero if the timer is disabled.

Each interrupt call takes approximately 50 cycles on entrance and 50 cycles on return. If `TPERIOD` and `TCOUNT` registers are set too low, you may incur an initializing overhead that could create an infinite loop.

Error Conditions

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

Example

```
#include <21160.h>
if (timer_set (1000, 1000) != 1)
    timer_on ();     /* enable timer */
```

See Also

timer_on, timer_off

The function is supplied only as an inlined procedure; that is, the compiler substitutes the appropriate statements for any reference to the procedure. Therefore, any source that references `timer_set` must include an appropriate header file.

transpm

matrix transpose

Synopsis

```
#include <matrix.h>

float *transpmf (float dm *output,
                 const float dm *a, int rows, int cols);

double *transpm (double dm *output,
                 const double dm *a, int rows, int cols);

long double *transpmd (long double dm *output,
                       const long double dm *a,
                       int rows, int cols);
```

Description

The transpm functions compute the linear algebraic transpose of the input matrix `a[][]`, and return the result in the matrix `output[][]`. The dimensions of these matrices are `a[rows][cols]`, and `output[cols][rows]`.

The algorithm for the linear algebraic transpose of a matrix is defined as:

$$c_{ji} = a_{ij}$$

The functions return a pointer to the output matrix.

Error Conditions

The transpm function do not return an error condition.

Example

```
#include <matrix.h>
#define ROWS 4
#define COLS 8

float a[ROWS][COLS];
float a_transpose[COLS][ROWS];

transpmf ((float *)(a_transpose),(float *)(a), ROWS, COLS);
```

See Also

No references to this function.

twidfftf

generate FFT twiddle factors for a fast FFT

Synopsis

```
#include <filter.h>
void twidfftf(float twid_real[], float twid_imag[], int n);
```

Description

The `twidfftf` function generates complex twiddle factors for the fast FFT function `cfftf`. The function calculates `+cosine` twiddle factors from `cos(0)` to `cos(pi)` and stores them in the vector `twid_real`, and calculates `-sine` factors from `-sin(0)` to `-sin(pi)` and stores them in the vector `twid_imag`.

The size of the vectors `twid_real` and `twid_imag` should be `n/2` where `n` must be a power of 2 and represents the size of the FFT.

For maximum efficiency, the `cfftf` function requires that the vectors `twid_real` and `twid_imag` are allocated in separate memory blocks.

Error Conditions

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

Example

```
#include <filter.h>

#define FFT_SIZE 1024

section("seg_dmdata") float twid_r[FFT_SIZE/2];
section("seg_pmdata") float twid_i[FFT_SIZE/2];

twidfftf(twid_r,twid_i,FFT_SIZE);
```

```
cfftf(input_r,input_i,
     temp_r,temp_i,
     twid_r,twid_i,FFT_SIZE);
```

See Also

cfftf

var

variance

Synopsis

```
#include <stats.h>

float varf (const float a[], int n);
double var (const double a[], int n);
long double vard (const long double a[], int n);
```

Description

These functions return the variance of the input array `a[]`. The length of the input array is `n`. The algorithm for computing the variance of a set of data is defined as:

$$c = \frac{n * \sum_{i=0}^{n-1} a_i^{\,2} - (\sum_{i=0}^{n-1} a_i)^2}{n(n-1)}$$

Error Conditions

The var functions do not return an error condition.

Example

```
#include <stats.h>
#define SIZE 256

double data[SIZE];
double result;

result = var (data, SIZE);
```

See Also

mean

By default, the `varf` function (and `var`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

vecdot

vector dot product

Synopsis

```
#include <vector.h>

float vecdotf (const float dm a[],
               const float dm b[], int samples);

double vecdot (const double dm a[],
               const double dm b[], int samples);

long double vecdotd (const long double dm a[],
                     const long double dm b[], int samples);
```

Description

The vecdot functions compute the dot product of the vectors `a[]` and `b[]`, which are `samples` in size. They return the scalar result.

The algorithm for calculating the dot product is:

$$return = \sum_{i=0}^{samples\ -1} a_i * b_i$$

where `i = {0,1,2,...,samples-1}`

Error Conditions

The vecdot functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double x[N], y[N];
```

```
double answer;

answer = vecdot (x, y, N);
```

See Also

cvecdot

By default, the `vecdotf` function (and `vecdot`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

vecsadd

vector + scalar addition

Synopsis

```
#include <vector.h>

float *vecsaddf (const float dm a[], float scalar,
                 float dm output[], int samples);

double *vecsadd (const double dm a[], double scalar,
                 double dm output[], int samples);

long double *vecsaddd (const long double dm a[],
                       long double scalar,
                       long double dm output[],
                       int samples);
```

Description

The vecsadd functions compute the sum of each element of the vector `a[]`, added to the scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecsadd functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input[N], result[N];
double x;

vecsadd (input, x, result, N);
```

See Also

cvecsadd

By default, the `vecsaddf` function (and `vecsadd`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

vecsmlt

vector * scalar multiplication

Synopsis

```
#include <vector.h>

float *vecsmltf (const float dm a[], float scalar,
                 float dm output[], int samples);

double *vecsmlt (const double dm a[], double scalar,
                 double dm output[], int samples);

long double *vecsmltd (const long double dm a[],
                       long double scalar,
                       long double dm output[],
                       int samples);
```

Description

The vecsmlt functions compute the product of each element of the vector `a[]`, multiplied by the scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecsmlt functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input[N], result[N];
double x;

vecsmlt (input, x, result, N);
```

See Also

cvecsmlt

By default, the `vecsmltf` function (and `vecsmlt`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

vecssub

vector - scalar subtraction

Synopsis

```
#include <vector.h>

float *vecssubf (const float dm a[], float scalar,
                 float dm output[], int samples);

double *vecssub (const double dm a[], double scalar,
                 double dm output[], int samples);

long double *vecssubd (const long double dm a[],
                       long double scalar,
                       long double dm output[],
                       int samples);
```

Description

The vecssub functions compute the difference of each element of the vector `a[]`, minus the scalar. Both the input and output vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecssub functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input[N], result[N];
double x;

vecssub (input, x, result, N);
```

See Also

cvecssub

By default, the `vecssubf` function (and `vecssub`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

vecvadd

vector + vector addition

Synopsis

```
#include <vector.h>

float *vecvaddf (const float dm a[], const float dm b[],
                 float dm output[], int samples);

double *vecvadd (const double dm a[], const double dm b[],
                 double dm output[], int samples);

long double *vecvaddd (const long double dm a[],
                       const long double dm b[],
                       long double dm output[],
                       int samples);
```

Description

The vecvadd functions compute the sum of each of the elements of the vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecvadd functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input_1[N];
double input_2[N], result[N];

vecvadd (input_1, input_2, result, N);
```

See Also

cvecvadd

By default, the `vecvaddf` function (and `vecvadd`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

vecvmlt

vector * vector multiplication

Synopsis

```
#include <vector.h>

float *vecvmltf (const float dm a[], const float dm b[],
                 float dm output[], int samples);

double *vecvmlt (const double dm a[], const double dm b[],
                 double dm output[], int samples);

long double *vecvmltd (const long double dm a[],
                       const long double dm b[],
                       long double dm output[],
                       int samples);
```

Description

The vecvmlt functions compute the product of each of the elements of the vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecvmlt functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input_1[N];
double input_2[N], result[N];

vecvmlt (input_1, input_2, result, N);
```

See Also

cvecvmlt

By default, the `vecvmltf` function (and `vecvmlt`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

vecvsub

vector - vector subtraction

Synopsis

```
#include <vector.h>

float *vecvsubf (const float dm a[], const float dm b[],
                 float dm output[], int samples);

double *vecvsub (const double dm a[], const double dm b[],
                 double dm output[], int samples);

long double *vecvsubd (const long double dm a[],
                       const long double dm b[],
                       long double dm output[],
                       int samples);
```

Description

The vecvsub functions compute the difference of each of the elements of the vectors `a[]` and `b[]`, and store the result in the output vector. All three vectors are `samples` in size. The functions return a pointer to the output vector.

Error Conditions

The vecvsub functions do not return an error condition.

Example

```
#include <vector.h>
#define N 100

double input_1[N];
double input_2[N], result[N];

vecvsub (input_1, input_2, result, N);
```

See Also

cvecvsub

By default, the `vecvsubf` function (and `vecvsub`, if `doubles` are the same size as `floats`) uses SIMD; refer to "Implications of Using SIMD Mode" on page 5-17 for more information.

zero_cross

count zero crossings

Synopsis

```
#include <stats.h>

int zero_crossf (const float in[], int length);
int zero_cross (const double in[], int length);
int zero_crossd (const long double in[], int length);
```

Description

The zero_cross functions return the number of times that a signal represented in the input array `in[]` crosses over the zero line. If all the input values are either positive or zero, or they are all either negative or zero, then the functions return a zero.

Error Conditions

The zero_cross functions do not return an error condition.

Example

```
#include <stats.h>
#define SIZE 256

double input[SIZE];
int result;

result = zero_cross (input, SIZE);
```

See Also

No references to this function.

I INDEX

Symbols

µ-law
  companders
    ADSP-2106x, 4-5
    ADSP-2116x/2126x/2136x DSPs, 5-6
  compression function
    ADSP-2106x, 4-116
    ADSP-2116x/2126x/2136x DSPs, 5-126
  expansion function
    ADSP-2106x, 4-117
    ADSP-2116x/2126x/2136x DSPs, 5-127

Numerics

21020.h header file, 4-4
21060.h header file, 4-4
21065L.h header file, 4-5
__2106x__ macro, 1-166
21160.h header file, 5-5
__2116x__ macro, 1-166
21262.h header file, 5-5
__2126x__ macro, 1-166
21363.h header file, 5-5
21364.h header file, 5-5
21365.h header file, 5-6
__2136x__ macro, 1-166
32-bit floating-point arithmetic, 1-62
32-bit IEEE single-precision format, 1-28
64-bit counter, 1-183
64-bit floating-point arithmetic, 1-62

A

-A (assert) compiler switch, 1-23
abend (see abort function)
abort (abnormal program end) function, 3-65
Abridged C++ library, 3-32
abs (absolute value, int) function, 3-66
absolute value (see abs, fabs, labs functions)
accessing
  system registers, 1-103
  system registers from C, 5-14
a_compress function, 4-16, 5-20
acos (arc cosine) functions, 3-67
add_devtab_entry function, 3-55
additional math functions
  average, 3-26
  clip, 3-26
  count bits set, 3-27
  maximum, 3-27
  multiple heaps, 3-27
__ADI_LIBEH__ macro, 1-59
__ADI_THREADS macro, 1-52
__ADSP21000__ macro, 1-45, 1-166
__ADSP21020__ macro, 1-167
__ADSP21060__ macro, 1-167
__ADSP21061__ macro, 1-167
__ADSP21062__ macro, 1-167
__ADSP21065L__ macro, 1-167
ADSP-21065L programmable timers
  disabling, 4-132
  enabling, 4-134

ADSP-2106x functions
a_compress, 4-16
a_expand, 4-17
alog, 4-18
alog10, 4-19
autocoh, 4-21
autocorr, 4-23
biquad, 4-25
cabs, 4-28
cadd, 4-29
cartesian, 4-30, 5-34
cdiv, 4-32, 5-36
cexp, 4-33
cfftN, 4-34
cmatmadd, 4-37
cmatmmlt, 4-39
cmatmsub, 4-41
cmatsadd, 4-43
cmatsmlt, 4-45
cmatssub, 4-47
cmlt, 4-49
conj, 4-50
convolve, 4-51
copysign, 4-53
cot, 4-54
crosscoh (cross-coherence), 4-55
crosscorr (cross-correlation), 4-57
csub (complex subtraction), 4-59
cvecdot, 4-60
cvecsadd, 4-62
cvecsmlt, 4-64
cvecssub, 4-66
cvecvadd, 4-68
cvecvmlt, 4-70
cvecvsub, 4-72
favg, 4-74
fclip, 4-75
fir, 4-76
fmax, 4-78
fmin, 4-79
fminf, 4-79, 5-88
histogram, 4-93
idle, 4-95
ifftN, 4-96
iir, 4-99
matinv, 4-102
matmadd (matrix addition), 4-103
matmmlt, 4-105
matsadd (matrix + scalar addition), 4-109
matsmlt, 4-111
matssub, 4-113
matsub, 4-107
mean, 4-115
mu_compress, 4-116
mu_expand, 4-117
norm, 4-118
polar, 4-119, 5-129
poll_flag_in, 4-121
rfftN, 4-123
rms, 4-126
rsqrt, 4-127
set_flag, 4-128
set_semaphore, 4-130
timer0_off, 4-132
timer0_on, 4-134
timer0_set, 4-137
timer1_off, 4-132
timer1_on, 4-134
timer1_set, 4-137
timer_off, 4-131
timer_on, 4-133
timer_set, 4-135
transpm, 4-139
var (variance), 4-141
vecdot, 4-142
vecsadd, 4-144
vecsmlt, 4-145
vecssub, 4-146
vecvadd, 4-147

vecvmlt, 4-149
vecvsub, 4-150
zero_cross, 4-152
ADSP-2106x processors
built-in functions, 4-13
DSP run-time library reference, 4-15 to 4-152
__ADSP21160__ macro, 1-167
ADSP-21160M processor
anomaly #40, 1-158
__ADSP21161__ macro, 1-167
ADSP-2116x/2126x/2136x functions
a_compress, 5-20
a_expand, 5-21
alog, 5-22
alog10, 5-23
arg, 5-24
autocoh, 5-25
autocorr, 5-27
biquad, 5-29
cabs, 5-32
cadd, 5-33
cexp, 5-37
cfftf, 5-43
cfft_mag, 5-38
cfftN, 5-40
cmatmadd, 5-46
cmatmmlt, 5-48
cmatmsub, 5-50
cmatsadd, 5-52
cmatsmlt, 5-54
cmatssub, 5-56
cmlt, 5-58
conj, 5-59
convolve, 5-60
copysign, 5-62
cot, 5-63
crosscoh, 5-64
crosscorr, 5-66
csub (complex subtraction), 5-68
cvecdot, 5-69
cvecsadd, 5-71
cvecsmlt, 5-73
cvecssub, 5-75
cvecvadd, 5-77
cvecvmlt, 5-79
cvecvsub, 5-81
favg, 5-83
fclip, 5-84
fir, 5-85
fmax, 5-87
fmin, 5-88
histogram, 5-102
idle, 5-104
ifftN, 5-105
iir, 5-108
matinv, 5-112
matmadd, 5-113
matmmlt, 5-115
matsadd, 5-119
matsmlt, 5-121
matssub, 5-123
matsub, 5-117
mean, 5-125
mu_compress, 5-126
mu_expand, 5-127
norm, 5-128
polar, 5-129
poll_flag_in, 5-131
rfft_mag, 5-133
rfftN, 5-135
rms, 5-138
rsqrt, 5-139
set_flag, 5-140
set_semaphore, 5-142
SIMD execution model, 5-41, 5-85, 5-106, 5-136
timer_off, 5-143
timer_on, 5-144
timer_set, 5-145

transpm, 5-147
twidfftf, 5-149
var, 5-151
vecdot, 5-153
vecsadd, 5-155
vecsmlt, 5-157
vecssub, 5-159
vecvadd, 5-161
vecvmlt, 5-163
vecvsub, 5-165
zero_cross, 5-167
ADSP-2116x/2126x/2136x processors
built-in functions, 5-15
DSP run-time library reference, 5-19
serial ports, 5-13
SIMD mode, 5-17
__ADSP21261__ macro, 1-168
__ADSP21262__ macro, 1-168
__ADSP21266__ macro, 1-168
__ADSP21267__ macro, 1-168
__ADSP21363__ macro, 1-168
__ADSP21364__ macro, 1-168
__ADSP21365__ macro, 1-168
__ADSP21366__ macro, 1-169
__ADSP21367__ macro, 1-169
__ADSP21368__ macro, 1-169
__ADSP21369__ macro, 1-169
a_expand function, 4-17, 5-21
aggregate assignment support (compiler), 1-101
aggregate constructor expression, 1-101
A-law
companders
ADSP-2106x, 4-5
ADSP-2116x/2126x/2136x DSPs, 5-6
compression function
ADSP-2106x, 4-16
in ADSP-21160 processor, 5-20
expansion function
ADSP-2106x, 4-17
in ADSP-21160 processor, 5-21
algebraic functions (see math functions)
algorithm header file, 3-37
alias
avoiding, 2-13
-aligned-stack (align stack) compiler switch, 1-24
alignment inquiry keyword, 1-145
__alignof__ (type-name) construct, 1-145
allocate memory (see calloc, free, malloc, realloc functions)
alphabetic character test (see isalpha function)
alphanumeric character test (see isalnum function)
alter macro, 1-216
alternate heap interface functions, 1-190
entry point names, 1-190
alternate registers, 1-192, 1-196
-alttok (alternative tokens) C++ mode compiler switch, 1-24
-anach (enable C++ anachronisms) compiler switch, 1-58
anachronisms
default C++ mode, 1-58
disabling in C++ mode, 1-60
__ANALOG_EXTENSIONS__ macro, 1-169
-annotate-loop-instr compiler switch, 1-25
annotations
assembly code, 2-51
assembly source code position, 2-58
loop identification, 2-54
modulo scheduling, 2-65, 2-66
vectorization, 2-62
ANSI standard compiler, 1-29
anti-log
base 10 functions, 4-19, 5-23
functions, 4-18, 5-22
archiver, 1-3

argc support, 1-185
arg (get phase of a complex number) functions, 4-20, 5-24
arguments and return transfer, 1-202
__argv_string variable, 1-185
argv support, 1-185
array
  zero length, 1-142
array search, binary (see bsearch function)
array storage, 1-205
ASCII string (*see* atof, atoi, atol, atold functions)
ASCII string (*see* atof, atoi, atol functions)
asctime (convert broken-down time into a string) function, 3-68
asctime function, 3-29, 3-106
asin (arc sine) functions, 3-70
asm
  construct template operands, 1-80
  keyword, 1-72, 1-75, 1-145
  statement, 1-143, 2-18
asm() construct, 1-75, 1-87
  flow control, 1-88
  multiple instructions, 1-86
  optimizing, 1-87
  reordering, 1-87
  syntax, 1-77
  syntax rules, 1-78
  template, 1-77
asm_sprt.h header file, 4-5, 5-6
asm_sprt.h system header file, 1-214
asm volatile() construct, 1-87
assembler
  for SHARC processors, 1-3
assembly
  code annotations, 2-51
  support keyword (asm), 1-227
assert.h header file, 3-13
atan2 (arc tangent division) functions, 3-72
atan (arc tangent) functions, 3-71
atexit (select exit function) function, 3-73
atof (convert string to double) function, 3-74
atoi (string to integer) function, 3-77
atold (convert string to long double) function, 3-79
atol (string to long integer) function, 3-78
__attribute__ keyword, 1-146
attributes
  functions, variables and types, 1-146
autocoh (autocoherence) functions, 4-21, 5-25
autocorr functions, 4-23, 5-27
automatic
  inlining, 1-41, 1-66, 2-17
  loop control variables, 2-28
  variables, 1-90
average (mean of 2 int) function, 3-82

B

background registers, 1-192, 1-196
base 10
  anti-log functions, 4-19, 5-23
basic cycle counting, 3-40
binary array search (see bsearch function)
bin_size parameter, 4-93, 5-102
biquad function, 4-25, 5-29
bit definitions
  processor-specific, 4-6, 5-8
bitfields
  signed, 1-50
  unsigned, 1-53
BITS_PER_WORD constant, 3-60
Boolean type support keywords (bool, true, false), 1-96
bool (see Boolean type support keywords (bool, true, false))
boot loader, 1-181
broken-down time, 3-27, 3-160, 3-201

bsearch (array search, binary) function, 3-83
-bss (placing data in bsz) compiler switch, 1-25
buf field, 3-62
-build-lib (build library) compiler switch, 1-25
build tools, 1-29
__builtin_aligned declaration, 2-7
__builtin_aligned function, 2-45
__builtin_aligned statement, 2-12
__builtin_circindex function, 2-34
__builtin_circptr function, 2-34, 2-35
built-in functions, 1-102
  ADSP-2106x processors, 4-13
  ADSP-2116x/2126x/2136x processors, 5-15
  C compiler, 3-30
  circular buffer, 1-107
  system, 2-32
bus lock
  controlling, 4-130, 5-142

C

C++
  gcc compatibility features not supported, 1-138
  member functions in assembly language, 1-220
  streambuf class, 3-39
  template inclusion control pragma, 1-130
  virtual lookup tables, 1-48
-c89 (ISO/IEC 9899
  1990 standard) compiler switch, 1-22
cabs (complex absolute value) functions, 4-28, 5-32
cadd (complex addition) functions, 4-29, 5-33
calendar time, 3-27, 3-230
calling
  assembly language subroutines from C/C++ programs, 1-209
  C/C++ functions from assembly language programs, 1-211
  C/C++ library functions, 3-3
  DSP library functions, 4-2, 5-2
calloc (allocate initialized memory) function, 3-85
calloc heap function, 1-185
call preserved registers, 1-194, 1-236
cartesian (Cartesian to polar) functions, 4-30, 5-34
cartesian number phase, 5-24
C/assembly interface (see mixed C/assembly programming)
ccall macro, 1-215, 1-228
C/C++ code optimization, 2-2
C/C++ compiler mode switches
  -c89, 1-22
C/C++ data types, 1-61
C/C++ language extensions, 1-71
  aggregate assignments, 1-73
  asm keyword, 1-75
  bool keyword, 1-72
  dm keyword, 1-90
  false keyword, 1-72
  indexed initializers, 1-73
  inline keyword, 1-74
  non-constant initializers, 1-73
  pm keyword, 1-90
  section keyword, 1-72
  true keyword, 1-72
  variable length arrays, 1-73
-c++ (C++ mode) compiler switch, 1-22
-C (comments) compiler switch, 1-25
-c (compile only) compiler switch, 1-26
C compiler
  benchmarking performance, 1-183

C/C++ run-time environment, 1-175
(see mixed C/C++/assembly programming)
C/C++ run-time libraries
thread-safe, 3-39
C/C++ run-time library, 3-4
ADSP-21020 and ADSP-2106x DSPs, 3-5
ADSP-2116x DSPs, 3-6, 3-7
ADSP-212xx DSPs, 3-8
ADSP-213xx DSPs, 3-10
C/C++ run-time library guide, 3-3 to 3-39
Cdef*.h header files, 4-7, 5-8
cdiv (complex division) functions, 4-32, 5-36
ceil (ceiling) functions, 3-86
cexp (complex exponential) functions, 4-33, 5-37
cfftf (fast N point complex input FFT) function, 5-43
cfft_mag function, 5-38
cfftN (N-point complex input fast Fourier transform) functions, 5-38, 5-40, 5-133
cfftN (N-point complex input FFT) functions, 4-34
C++ fractional arithmetic, 1-146
character string search, recursive (see strrchr function)
character string search (see strchr function)
char storage, 1-205
circindex (circular buffer operation on loop index) function, 3-87
circptr (circular buffer operation on pointer) function, 3-89
circular buffering
enabling, 1-195
circular buffer operation
on a pointer, 3-89
on loop index, 3-87
circular buffers, 2-33
built-in functions, 1-107
increment of array references, 1-108
increment of index, 1-107
increment of pointer, 1-107
interrupt dispatcher, 3-18
setting features of, 4-8, 5-11
switch setting, 1-30
C++ language extension
fract data type, 1-73
C language extensions
C++ style comments, 1-73
preprocessor generated warnings, 1-73
clear_interrupt (clear pending) function, 3-92
clip (x by y, int) function, 3-101
clobbered
registers, 1-123, 1-124
register sets, 1-125
clock (processor time) function, 3-45, 3-48, 3-102
CLOCKS_PER_SEC macro, 3-27, 3-45, 3-47
clock_t data type, 3-27, 3-45, 3-102
close function, 3-52
cmatmadd (complex matrix + matrix addition) functions, 4-37, 5-46
cmatmmlt (complex matrix * matrix multiplication) functions, 4-39
cmatmmlt (complex matrix matrix multiplication) functions, 5-48
cmatmsub (complex matrix - matrix subtraction) functions, 4-41, 5-50
cmatrix.h header file, 4-5, 5-6
cmatsadd (complex matrix + scalar addition) functions, 4-43
cmatsadd (complex matrix scalar addition) functions, 5-52
cmatsmlt (complex matrix * scalar multiplication) function, 4-45

cmatsmlt (complex matrix scalar multiplication) function, 5-54
cmatssub (complex matrix - scalar subtraction) functions, 4-47
cmatssub (complex matrix scalar subtraction) functions, 5-56
cmlt (complex multiplication) functions, 5-58
cmlt (complex multiply) functions, 4-49
C++ mode compiler switches
  -anach (enable C++ anachronisms), 1-58
  -eh (enable exception handling), 1-59
  -no-anach (disable C++ anachronisms), 1-60
  -no-eh (disable exception handling), 1-60
  -no-implicit-inclusion, 1-60
  -no-rtti (disable run-time type identification), 1-60
  -rtti (enable run-time type identification), 1-61
code generation pragmas
  #pragma avoid_anomaly_45, 1-138
code optimization
  enabling, 1-40
  for size, 1-41
comm.h header file, 4-5, 5-6
compare, strings (see strcmp, strcoll, strcspn, strpbrk, strncmp, strstr functions)
compare memory range (see memcmp function)
-compatible-pm-dm compiler switch, 1-26
compiler
  C/C++ language extensions, 1-70
  code optimization, 1-64, 2-2
  command-line interface, 1-4
  optimizer, 2-4
  prelinker, 1-68
  registers, 1-192
complex
  addition functions, 4-29, 5-33
  conjugate function, 4-50, 5-59
  division functions, 4-32, 5-36
  matrix addition functions, 4-37
  matrix functions, 4-5, 5-6
  matrix matrix addition functions, 5-46
  matrix matrix multiplication function, 4-39
  matrix matrix multiplication functions, 5-48
  matrix matrix subtraction function, 4-41, 5-50
  matrix scalar addition function, 4-43, 5-52
  matrix scalar multiplication function, 5-54
  matrix scalar multiplication functions, 4-45
  matrix scalar subtraction function, 4-47, 5-56
  multiplication functions, 4-49, 5-58
  number (phase of), 4-20, 5-24
  subtraction functions, 4-59, 5-68
  vector functions, 4-6, 5-7
complex_float
  operator, 3-32
complex header file, 3-32
complex.h header file (ADSP-2106x), 4-5
complex.h header file (ADSP-2116x/2126x/2136x), 5-7
complex_long_double operator, 3-32
compound statement., 1-172
concatenate, string (see strcat, strncat function)
conditional
  code in loops, 2-26
  expressions with missing operands, 1-141

conj (complex conjugate) functions, 4-50, 5-59
const
  keyword, 2-30
  pointers, 1-26
  qualifier, 2-30
-const-read-write compiler switch, 1-26
-const-read-write flag, 2-30
construct
  from polar coordinates (polar function), 4-119, 5-129
constructs
  flow control, 1-88
  input and output operands, 1-87
continuation characters, 1-36, 1-39
control character test (see iscntrl function)
convert, characters (see tolower, toupper functions)
convert, strings to long integer (see atof, atoi, atol, strtok, strtol, strtoul, functions)
convolve (convolution) function, 4-51, 5-60
copy, string (see strcpy, strncpy function)
copy memory range (see memcpy function)
copysign functions, 4-53, 5-62
cos (cosine) functions, 3-103
cosh (hyperbolic cosine) functions, 3-104
cot (cotangent) functions, 4-54, 5-63
count_ones function, 3-105
__cplusplus macro, 1-169
C++ programming examples, 1-223
  complex support, 1-224
  fract support, 1-223
crosscoh (cross-coherence) functions, 4-55, 5-64
crosscorr (cross-correlation) functions, 4-57, 5-66
C run-time library reference, 3-64 to 3-236
C++ style comments, 1-102
csub (complex subtraction) functions, 4-59, 5-68
ctime (convert calendar time into a string) function, 3-68, 3-106
C type functions
  islower, 3-139
C-type functions
  isalnum, 3-134
  isalpha, 3-135
  iscntrl, 3-136
  isdigit, 3-137
  isgraph, 3-138
  islower, 3-141
  isprint, 3-144
  ispunct, 3-145
  isspace, 3-146
  isupper, 3-147
  isxdigit, 3-148
  tolower, 3-231
  toupper, 3-232
ctype.h header file, 3-13
cvecdot (complex vector dot product) functions, 4-60, 5-69
cvecsadd (complex vector scalar addition) functions, 4-62, 5-71
cvecsmlt (complex vector scalar multiplication) functions, 4-64, 5-73
cvecssub (complex vector scalar subtraction) functions, 4-66, 5-75
cvector.h header file, 4-6, 5-7
cvecvadd (complex vector addition) functions, 4-68, 5-77
cvecvmlt (complex vector multiplication) functions, 4-70, 5-79
cvecvsub (complex vector subtraction) functions, 4-72, 5-81
cycle_count.h header file, 3-13, 3-40, 3-41
cycle count register, 3-41, 3-42, 3-48
cycle counts, 3-13
CYCLE_COUNT_START macro, 1-184

CYCLE_COUNT_STOP macro, 1-184
cycles.h header file, 3-13, 3-29, 3-42, 3-43
CYCLES_INIT(S) macro, 3-42
CYCLES_PRINT(S) macro, 3-43
CYCLES_RESET(S) macro, 3-43
CYCLES_START(S) macro, 3-42
CYCLES_STOP(S) macro, 3-42
cycle_t data type, 3-41

D

data
  fetching with 32-bit loads, 2-11
  field, 3-51
  memory storage, 1-179
  packing, 3-60
  storage formats, 1-205
  word alignment, 2-11
data alignment pragmas, 1-111
data_imag array, 5-43
data_real array, 5-43
data types, 1-61
  double, 1-62
  float, 1-62
  fract, 1-62, 1-146
  int, 1-62
  long, 1-62
  long int, 1-62
  scalar, 2-8
__DATE__ macro, 1-169
Daylight Saving flag, 3-27
-DCLOCKS_PER_SEC= compile-time switch, 3-47
-D (define macro) compiler switch, 1-26, 1-53
-DDO_CYCLE_COUNTS compile-time switch, 3-42, 3-48
-DDO_CYCLE_COUNTS switch, 3-41
deallocate memory (see free function)
debugging
  sourcr-level, 1-31
debugging information, 1-31
  for header file, 1-26
  removing, 1-48
-debug-types compiler switch, 1-26
declarations
  mixed with code, 1-144
default
  device, 3-58
  sections, 1-133
  target processor, 1-45
-default-linkage (assembler, C, or C++) compiler switch, 1-27
default run-time header files, 1-182
#define preprocessor command, 1-172
defining
  run-time label, 3-184
delayed branches, 1-38
deque header file, 3-37
DevEntry structure, 3-50
device
  default, 3-58
  driver, 3-50
  drivers, 3-50
  identifiers, 3-50
  pre-registering, 3-56
device.h header file, 3-14, 3-50
DeviceID field, 3-51
device_int.h header file, 3-14
devtab.c library source file, 3-56
diagnostics
  control pragma, 1-135
difftime (difference between two calendar times) function, 3-107
digit character test (see isdigit function)
disabling
  ADSP-21065L programmable timers, 4-132
div (division, int) function, 3-108
division
  complex, 4-32, 5-36

division (see div, ldiv functions)
dma.h header file, 4-7, 5-9
DMA support functions, 5-9
dm (see dual memory support keywords (pm dm))
double
  representation, 3-213
  storage format, 1-27, 1-205
DOUBLE32 qualifier, 1-134
DOUBLE64 qualifier, 1-134
DOUBLEANY qualifier, 1-134
__DOUBLES_ARE_FLOATS__ macro, 1-28, 1-170
-double-size-32 (single-precision double) compiler switch, 1-27
-double-size-64 (double-precision double) compiler switch, 1-27
-double-size-any compiler switch, 1-28
driver I/O redirection, 1-56
-dry-run (verbose dry-run) compiler switch, 1-28
-dry (terse -dry-run) compiler switch, 1-28
DSP run-time
  library calls, 5-2
  library routines, 5-2
  linking programs, 5-3
dual compute-block architectures, 2-11
dual-memory support keywords (pm dm), 1-90
dual-word-aligned addresses, 2-11
dual-word boundary, 2-13
dynamic_cast run-time type identification, 1-60

E

easm21k utility, 1-3
__ECC__ macro, 1-170
__EDG__ macro, 1-170
__EDG_VERSION__ macro, 1-170
EDOM macro, 3-17
-ED (run after preprocessing to file) compiler switch, 1-29
-EE (run after preprocessing) compiler switch, 1-29
-eh (enable exception handling) compiler switch, 1-59
elfar archive library, 1-3
elfloader utility, 1-180, 1-181
Embedded C++ library header files
  complex, 3-32
  exception, 3-33
  fract, 3-33
  fstream, 3-33
  fstreams.h, 3-39
  iomanip, 3-33
  ios, 3-33
  iosfwd, 3-34
  iostream, 3-34
  iostream.h, 3-39
  istream, 3-34
  new, 3-34
  new.h, 3-39
  ostream, 3-34
  sstream, 3-34
  stdexcept, 3-34
  streambuf, 3-35
  string, 3-35
  strstream, 3-35
Embedded Standard Template Library, 3-37
EMUCLK2 register, 1-183
EMUCLK register, 1-183, 3-42, 3-49
emulated arithmetic
  avoiding, 2-9
enabling
  ADSP-21065L programmable timers, 4-134
  optimization and debugging, 1-41
end (see atexit, exit functions)
entry macro, 1-214

environment variables, 1-63
ADI_DSP, 1-64
CC21K_IGNORE_ENV, 1-64
CC21K_OPTIONS, 1-64
PATH, 1-63
TEMP, 1-64
TMP, 1-63
ERANGE macro, 3-17
errno global variable, 3-29
errno.h header file, 3-14
error messages
control pragma, 1-135
escape character, 1-145
-E (stop after preprocessing) compiler switch, 1-29
examples
inline assembly (add), 1-227
registers for arguments and return (add 2), 1-233
scratch registers (dot oroduct), 1-229
stack for arguments and return (add 5), 1-232
void functions (delay), 1-231
exception handler
disabling, 1-60
enabling, 1-59
exception header file, 3-33
__EXCEPTIONS macro, 1-59
exit macro, 1-200, 1-214
exit (program termination) function, 3-109
expected_false built-in function, 1-108, 2-21
expected_true built-in function, 1-108, 2-21
exp (exponential) functions, 3-110
exponential (see exp, ldexp functions)
exponentiation, 4-18, 4-19, 5-22, 5-23
.EXTERN assembler directive, 1-218
-extra-keywords (not quite -analog) compiler switch, 1-29
EZ-KIT Lite system, 3-14, 3-24, 3-50, 3-58

F

fabs (absolute value) functions, 3-111
false (see Boolean type support keywords (bool, true, false))
far jump return (see longjmp, setjmp functions)
Fast Fourier Transform (FFT) functions, 4-10, 5-9, 5-14
fast hardware floating-point instructions, 1-28
fast interrupt dispatcher, 3-20
fast N-point complex input FFT (cfftf) function, 5-43
favg (mean of two values) functions, 4-74, 5-83
fclip functions, 4-75, 5-84
FFT twiddle factors for fast FFT, 5-149
file
annotation position, 2-58
extensions, 1-5, 1-7
name (description), 1-23
searches, 1-7
fileID field, 3-63
file I/O
extending to new devices, 3-50
file I/O support, 3-49
__FILE__ macro, 1-170
-@ filename (command file) compiler switch, 1-23
fill memory range (see memset function)
filter.h header file, 5-9
filters
for ADSP-2106x processors, 4-8, 5-11
for ADSP-2116x/2126x/2136x processors, 5-9, 5-11
signal processing, 5-9
filters.h header file, 4-8, 5-11

final interrupt dispatcher, 3-21
finish processing argument list (see va_end function)
finite impulse response (FIR) filter, 4-76, 5-85
fir function, 4-76, 5-85
flags, 4-128
-flags (command line input) compiler switch, 1-29
flags field, 3-61
float.h header file, 3-15
floating-point
  data types, 1-62
  hexadecimal constants, 1-142
floating-point underflow
  avoiding, 1-30
float storage format, 1-27, 1-205
-float-to-int compiler switch, 1-30
floor (integral value) functions, 3-112
flow control operations, 1-88
FLT_MAX macro, 3-15
FLT_MIN macro, 3-15
fmax (maximum) functions, 4-78, 5-87
fmin (minimum) functions, 4-79, 5-88
fmod (floating-point modulus) functions, 3-113
fopen() function, 3-58
-force-circbuf (circular buffer) compiler switch, 1-30
-force-circbuf switch, 2-34
-fp-associative (floating-point associative operation) compiler switch, 1-30
fract data type, 1-62, 1-146
fract header file, 3-33
fractional
  arithmetic operators, 1-148
  (fixed-point) arithmetic, 1-146
  literals, 1-147
  saturated arithmetic, 1-149
frame pointer, 1-196, 1-197
free (deallocate memory) functions, 3-114
free heap function, 1-185
frexp (fraction/exponent) functions, 3-115
fstream header file, 3-33
fstream.h header file, 3-39
-full-version (display versions) compiler switch, 1-30
function
  arguments/return value transfer, 1-202
  call return address, 1-228
  declarations with pointers, 1-93
  entry (prologue), 1-197, 1-228
  exit (epilogue), 1-197, 1-228
  inlining, 2-16
  primitive I/O, 3-24
functional header file, 3-37
functions
  calling in loop, 2-27

G

GCC compatibility extensions, 1-138
GCC compatibility mode, 1-138
gen_bartlett (generate bartlett window) function, 4-80, 5-89
gen_blackman (generate blackman window) function, 4-82, 5-91
general optimization pragmas, 1-119
gen_gaussian (generate gaussian window) function, 4-83, 5-92
gen_hamming (generate hamming window) function, 4-84, 5-93
gen_hanning (generate hanning window) function, 4-85, 5-94
gen_harris (generate harris window) function, 4-86, 5-95
gen_kaiser (generate kaiser window) function, 4-87, 5-96
gen_rectangular (generate rectangular window) function, 4-89, 5-98

gen_triangle (generate triangle window) function, 4-90, 5-99
gen_vohann (generate von hann window) function, 4-92, 5-101
get_default_io_device, 3-57
getenv (get string definition from operating system) function, 3-116
get locale pointer (see localeconv function)
get next argument in list (see va_arg function)
gets macro, 1-215
-g (generate debug information) compiler switch, 1-31
globvar global variable, 2-29
gmtime (convert calendar time into broken-down time as UTC) function, 3-117
gmtime function, 3-29, 3-68, 3-160
GNU C compiler, 1-138
graphical character test (see isgraph function)
__GROUPNAME__ macro, 1-52

H

hardware
  defect workarounds, 1-56
hash_map header file, 3-37
hash_set header file, 3-37
header
  stop point, 1-129
header file control pragmas, 1-129
header files
  C run-time library
    float.h, 3-15
    iso646.h, 3-15
  def21020.h, 4-7
  def21060.h, 4-7
  def21061.h, 4-7
  def21062.h, 4-7
  def21065L.h, 4-7
  def21161.h, 5-8
  def21261.h, 5-8
  def21262.h, 5-8
  def21266.h, 5-8
  def21267.h, 5-8
  def21363.h, 5-8
  def21364.h, 5-8
  def21365.h, 5-8
  def21366.h, 5-8
  def21367.h, 5-8
  def21368.h, 5-8
  def21369.h, 5-8
  defining processor-specific symbolic names, 4-6, 5-8
  system, 1-214
  working with, 3-12
header files (ADSP-2106x), 4-4
  21020.h, 4-4
  21060.h, 4-4
  21065l.h, 4-5
  asm_sprt.h, 4-5
  cmatrix.h, 4-5
  comm.h, 4-5
  complex.h, 4-5
  cvector.h, 4-6
  dma.h, 4-7
  filters.h, 4-8
  macros.h, 4-8
  math.h, 4-8
  matrix.h, 4-9
  saturate.h, 4-10
  sprt.h, 4-10
  stats.h, 4-10
  sysreg.h, 4-10
  trans.h, 4-10
  vector.h, 4-11
  window.h, 4-11
header files (ADSP-2116x/2126x/2136x), 5-5
  21160.h, 5-5

21262.h, 5-5
21363.h, 5-5
21364.h, 5-5
21365.h, 5-6
asm_sprt.h, 5-6
cmatrix.h, 5-6
comm.h, 5-6
complex.h, 5-7
cvector.h, 5-7
dma.h, 5-9
filter.h, 5-9
filters.h, 5-11
macro.h, 5-11
math.h, 5-11
matrix.h, 5-13
saturate.h, 5-13
sprt.h, 5-13
stats.h, 5-13
sysreg.h, 5-14
trans.h, 5-14
vector.h, 5-14
window.h, 5-14
header files (C++ for C facilities)
cassert, 3-36
cctype, 3-36
cerrno, 3-36
cfloat, 3-36
climits, 3-36
clocale, 3-36
cmath, 3-36
csetjmp, 3-36
csignal, 3-36
cstdarg, 3-36
cstddef, 3-36
cstdio, 3-36
cstdlib, 3-36
cstring, 3-36
header files (standard)
device.h, 3-14
device_int.h, 3-14
iso646.h, 3-15
heap, 3-118
allocating and initializing memory in, 3-118
allocating memory from, 3-127
allocating uninitialized memory, 3-166
alternate, 1-189, 1-190
changing memory allocation from, 3-129
identifier, 1-189
obtaining primary heap identifier, 3-125
return memory to, 3-120
setting for dynamic memory allocation, 3-186
standard functions, 1-185
heap_calloc function, 1-185, 1-190, 3-118
heap_free function, 1-185, 1-190, 3-120
heap_install function, 3-122
heap_lookup_name function, 3-122, 3-125
heap_malloc function, 1-185, 1-190, 3-127
heap_realloc function, 1-185, 1-190, 3-129
heaps
multiple, 1-185
heap_switch function, 1-189
-help (command-line help) compiler switch, 1-32
hexadecimal digit test (see isxdigit function)
hexadecimal floating-point constants, 1-142
-HH (list *.h and compile) compiler switch, 1-31
histogram function, 4-93, 5-102
-H (list *.h) compiler switch, 1-31
__HOSTNAME__ macro, 1-52
HUGE_VAL macro, 3-17
hyperbolic (see cosh, sinh, tanh functions)

I

IDDE_ARGS macro, 1-185
idle function, 4-95, 5-104
IEEE single/double precision formats, 1-205
ifftN (N-point radix-2 inverse Fast Fourier transform) functions, 4-96, 5-105
-I (include search directory) compiler switch, 1-40
iir (infinite impulse response) function, 4-99, 5-108
-i (less includes) compiler switch, 1-33
implicit inclusion, 1-130
include files, 1-32
-include (include file) compiler switch, 1-33
indexed
  array, 2-15
  initializer support (compiler), 1-100
index in a loop, 3-87
induction variables, 2-25
infinite impulse response (IIR) filter, 5-108
init function, 3-51
initialization
  data storage, 1-180
  section processing, 1-180
initialize argument list (see va_start function)
initializer
  indexed, 1-99
  non-constant, 1-99
initializing
  DSP timer, 5-145
initiation interval, 2-65
inline
  asm statements, 2-18
  avoiding code, 2-37
  keyword, 1-74, 2-17, 2-37
inline assembly language support keyword (asm), 1-75
  construct optimization, 1-87
  constructs with multiple instructions, 1-86
  construct template, 1-77
inline function support keyword (inline), 1-72, 1-74
inlining
  automatic, 2-17
  file position, 2-58
  function, 2-16
inner loop, 2-26
input flag
  testing, 4-121, 5-131
input operand, 1-87
instantiation
  template functions, 1-127
integer data types, 1-62
interface support macros, 1-218
  alter, 1-216
  ccall, 1-215
  C/C++ and assembly, 1-214
  entry, 1-214
  exit, 1-214
  gets, 1-215
  leaf_entry, 1-215
  leaf_exit, 1-215
  puts, 1-215
  reads, 1-215
  restore_reg, 1-216
  save_reg, 1-216
interfacing C/C++ and assembly (see mixed C/C++/assembly programming)
intermediate files
  saving, 1-48
interprocedural analysis, 1-67
interprocedural analysis (IPA), 2-7

interrupt
  length, 1-183
  pragma, 1-115
  table, 1-208
  vector table area, 1-182
interrupt dispatchers, 3-18
  circular buffer, 3-18, 3-21
  fast, 3-20
  final, 3-21
  normal, 3-19
  pragma, 3-21
  super-fast, 1-197, 3-20
interrupt enable bit, 3-22
interruptf() function, 3-20
interruptfnsm() function, 3-22
interrupt handler, 1-116
interrupt (interrupt handling) function, 3-132
interrupt nesting, 3-19
  disabling, 3-20
  restrictions with ADSP-2116x/2126x/2136x DSPs, 3-22
interrupt-safe functions, 3-29
interrupts (see clear_interrupt, interruptf, interrupts, signal, raise functions)
interruptss() function, 3-21
intrinsic (built-in) functions, 1-102
int storage format, 1-205
inverse (see acos, asin, atan, atan2 functions)
I/O
  extending to new devices, 3-50
  functions, 3-24
  primitives, 3-50, 3-58
  support for new devices, 3-50
iomanip header file, 3-33
iomanip.h header file, 3-39
I/O primitives, 3-49
  data packing, 3-59
  data structure, 3-60
iosfwd header file, 3-34
ios header file, 3-33
iostream header file, 3-34
iostream.h header file, 3-39
IPA, 1-67
-ipa (interprocedural analysis) compiler switch, 1-33, 2-7
IRPTEN interrupt enable bit, 3-22
isalnum (alphanumeric character test) function, 3-134
isalpha (alphabetic character test) function, 3-135
iscntrl (control character test) function, 3-136
isdigit (digit character test) function, 3-137
isgraph (graphical character test) function, 3-138
isinf (test for infinity) function, 3-139
islower (lower case character test) function, 3-141
isnan (test for NAN) function, 3-142
iso646.h (Boolean operator) header file, 3-15
isprint (printable character test) function, 3-144
ispunct (punctuation character test) function, 3-145
isspace (white space character test) function, 3-146
-I (start include directory) compiler switch, 1-32
istream header file, 3-34
isupper (uppercase character test) function, 3-147
isxdigit (hexadecimal digit test) function, 3-148
iterator header file, 3-37

K

keywords (compiler) (see compiler C/C++ extensions)

L

labs (absolute value, long) function, 3-149
language extensions (compiler) (see compiler C/C++ extensions)
lavg (mean of two values) function, 3-150
LC_COLLATE macro, 3-197
lclip function, 3-151
lcount_ones function, 3-152
ldexp (exponential, multiply) functions, 3-153
ldiv (division, long) function, 3-154
leaf assembly routines, 1-226
leaf_entry macro, 1-215
leaf_exit macro, 1-200, 1-215
legacy code, 1-95
___lib_prog_term label, 3-109
library source code
  working with, 4-3, 5-4
library source file
  devtab.c, 3-56
__lib_setup_processor routine, 1-182
limits.h header file, 3-16
line breaks
  in string literals, 1-143
__LINE__ macro, 1-170
linking
  DSP library functions (ADSP-2106x DSPs), 4-3
  DSP library functions (ADSP-2116x/2126x/2136x), 5-3
  pragmas for, 1-132
list header file, 3-37
-L (library search directory) compiler switch, 1-34
-l (link library) compiler switch, 1-34
lmax (maximum) function, 3-155
lmin (minimum) function, 3-156
localeconv (localization pointer) function, 3-157
locale.h header file, 3-16
localization (see localeconv, setlocale, strxfrm functions)
localtime (convert calendar time into broken-down time) function, 3-160
localtime function, 3-29, 3-68, 3-117
log10 (log base 10) functions, 3-163
log (log base e) functions, 3-162
long
  latencies, 2-31
  storage format, 1-205
long double
  representation, 3-216
longjmp (far jump return) function, 3-164
long jump (see longjmp, setjmp functions)
loop
  annotations, 2-66
  control
    variables, 2-28
  cycle count, 2-56
  exit test, 2-28
  flattening, 2-61
  identification annotation, 2-54, 2-55
  iteration count, 2-43
  optimization, 1-116, 2-43
  parallel processing, 1-118
  resource usage, 2-56
  rotation by hand., 2-24
  scheduled, 2-65
  short, 2-22
  trip count, 2-60
  unrolling, 2-22
  vectorization, 2-44
  vectorizing, 1-116
loop-carried dependency, 2-23, 2-24
lowercase (see islower, tolower functions)

L registers, 1-193
L-registers, 1-125
-L (search library) compiler switch, 1-40
lvalue
  GCC generalized, 1-141
  generalized, 1-141

M

__MACHINE__ macro, 1-52
macro.h header file, 4-8, 5-11
macros
  ccall, 1-228
  compound statements as, 1-172
  EDOM, 3-17
  ERANGE, 3-17
  __GROUPNAME__, 1-52
  __HOSTNAME__, 1-52
  HUGE_VAL, 3-17
  IDDE_ARGS, 1-185
  interface support, 1-218
  LC_COLLATE, 3-197
  __MACHINE__, 1-52
  mixed C/assembly support, 1-214
  predefined, 1-166
  __REALNAME__, 1-52
  __RTTI, 1-61
  SKIP_SPACES, 1-172
  stack management, 1-228
  __SYSTEM__, 1-52
  __USERNAME__, 1-52
  variable argument, 1-143
malloc (allocate uninitialized memory) function, 3-166
malloc heap function, 1-185
managing
  stacks, 1-197
-map (generate a memory map) compiler switch, 1-36
map header file, 3-38
math functions
  acos, 3-67
  additional, 4-8, 5-11
  asin, 3-70
  atan, 3-71
  atan2, 3-72
  ceil, 3-86
  ceil, ceilf, 3-89
  cos, 3-103
  cosh, 3-104
  exp, 3-110
  fabs, 3-111
  floor, 3-112
  fmod, 3-113
  frexp, 3-115
  ldexp, 3-153
  log, 3-162
  log10, 3-163
  modf, 3-176
  pow, 3-177
  rsqrt, 4-127, 5-139
  sin, sinf, 3-190
  sinh, 3-191
  sin (sine), 3-190
  sqrt, 3-192
  standard, 4-8, 5-11
  tan, 3-228
  tanh, 3-229
math.h header file, 3-16, 4-8, 5-11
matinv (real matrix inversion) functions, 4-102, 5-112
matmadd (matrix addition) functions, 4-103, 5-113
matmmlt (matrix multiplication) functions, 4-105, 5-115
matmsub (matrix subtraction) functions, 4-107, 5-117
matrix addition functions, 4-103, 5-113
matrix.h header file, 4-9, 5-13

matrix scalar addition functions, 4-109, 5-119
matrix transpose, 4-139, 5-147
matsmlt (real matrix scalar multiplication) function, 4-111
matsmlt (real matrix scalar multiplication) functions, 5-121
matssub (real matrix scalar subtraction) function, 4-113, 5-123
matsub (matrix subtraction) function, 4-107, 5-117
maximum performance., 2-36
max (maximum) function, 3-167
-MD (make and compile) compiler switch, 1-35
mean functions, 4-115, 5-125
mem21k initializer
  disabling, 1-39
mem21k utility, 1-180, 1-181
memchr (find character) function, 3-168
memcmp (compare memory range) function, 3-169
memcpy (copy memory range) function, 3-170
-mem (enable memory initialization) compiler switch, 1-36
memmove (move memory range) function, 3-171
memory
  data placement in, 2-19
  header file, 3-38
  initializer (mem21k), 1-181
  map file, 1-36
  placing code in, 1-177
  section names, 1-177
  used for placing code in, 1-177
memory functions (see calloc, free, malloc, memcmp, memcpy, memset, memmove, memchar, realloc functions)
memory-mapped registers
  accessing, 4-7, 5-8
memset (fill memory range) function, 3-172
minimum code size, 2-36
min (minimum) function, 3-173
missing operands
  in conditional expressions, 1-141
mixed C/assembly naming conventions, 1-218
mixed C/assembly programming
  arguments and return, 1-202
  asm() constructs, 1-75, 1-77, 1-80, 1-86
  call preserved registers, 1-194
  compiler registers, 1-192
  data storage and type sizes, 1-205
  examples, 1-226
  return address, 1-228
  scratch registers, 1-195
  stack registers, 1-196
  stack usage, 1-197
  user registers, 1-193
mixed C/assembly support, 4-5
  macros, 1-214
mixed C/C++/assembly programming, 1-209
mktime (convert broken-down time into a calendar) function, 3-174
-M (make only) compiler switch, 1-35
MMASK register, 1-192
-MM (make rules and compile) compiler switch, 1-35
MODE2 register, 4-121, 5-131, 5-140
  setting, 4-128
modf (modulus, float) functions, 3-176
modulo scheduling, 2-65
  annotations, 2-66
modulo variable expansion unroll, 2-65
modulus array references, 1-108

-Mo (processor output file) compiler switch, 1-35
move memory range (see memmove function)
-MQ (output without compile) compiler switch, 1-36
M_STRLEN_PROVIDED bit, 3-62
-Mt filename (output make rule) compiler switch, 1-35
mu_compress function, 4-116, 5-126
mu_expand function, 4-117, 5-127
multi-line asm() C program constructs, 1-86
-multiline compiler switch, 1-36
multiple heap, 3-122
multiple heaps, 1-185

N

naming conventions
  assembly and C, 1-219
  assembly and C/C ++, 1-218
natural logarithm (see log functions)
nCompleted field, 3-63
nDesired field, 3-62
nested hardware loops, 1-47
new devices
  I/O support, 3-50
  registering, 3-55
new header file, 3-34
new.h header file, 3-39
newline
  in string literals, 1-36, 1-39
-no-aligned-stack (do not align stack) compiler switch, 1-36
-no-alttok (disable tokens) C++ mode compiler switch, 1-36
-no-anach (disable C++ anachronisms) compiler switch, 1-60
-no-annotate (disable assembly annotations) compiler common switch, 1-37
-no-annotate-loop-instr compiler common switch, 1-37
-no-bss compiler common switch, 1-37
__NO_BUILTIN macro, 1-171
-no-builtin (no built-in functions) compiler switch, 1-37
__NO_BUILTIN preprocessor macro, 1-37
-no-circbuf (no circular buffer) compiler switch, 1-38
-no-db (no delayed branches) compiler switch, 1-38
-no-def (disable definitions) compiler switch, 1-38
-no-demangle compiler switch, 1-60
-no-eh (disable exception handling) C++ mode compiler switch, 1-60
-no-extra-keywords (not quite -ansi) compiler switch, 1-38
-no-fp-associative compiler switch, 1-38
no implicit inclusion, 1-130
-no-implicit-inclusion C++ mode compiler switch, 1-60
NO_INIT qualifier, 1-134
_NO_LONGLONG macro, 1-170
-no-mem (disable memory initialization) compiler switch, 1-39
-no-multiline compiler switch, 1-39
non-constant initializer support (compiler), 1-99
non-leaf
  assembly routines, 1-226
  routines to make calls (RMS), 1-234
non-unit stride
  avoiding, 2-27
normal interrupt dispatcher, 3-19
normalized fraction (see frexp functions)

norm (normalization) functions, 4-118, 5-128
-no-rtti (disable run-time type identification) C++ mode compiler switch, 1-60
-no-saturation (no faster operations) compiler switch, 1-39
-no-simd (disable SIMD mode) compiler switch, 1-39
-no-std-ass (disable standard assertions) compiler switch, 1-39
-no-std-def (disable standard definitions) compiler switch, 1-40
-no-std-inc (disable standard include search) compiler switch, 1-40
-no-std-lib (disable standard library search) compiler switch, 1-40
Not a Number (NaN) test, 3-142
-no-threads (disable thread-safe build) compiler switch, 1-40
N-point complex input FFT functions, 4-34, 5-40
N-point inverse FFT functions, 4-96, 5-105
N-point real input FFT functions, 4-123, 5-135
numeric header file, 3-38

O

-Oa (automatic function inlining) compiler switch, 1-41
objects
  copy characters between overlapping, 3-171
-O (enable optimization) compiler switch, 1-40
-Og (optimize while preserving debugging information) compiler switch, 1-41
-o (output) compiler switch, 1-42
open function, 3-51
open() function, 3-58
operand constraints, 1-81
optimization
  code, 2-36
  controlling, 1-64
  default, 1-65
  enabling, 1-40
  reporting progress, 1-45, 1-46
  reporting progress in, 1-45, 1-46
  switches, 2-47
  turning on, 1-67
optimization and debugging performance, 1-41
optimizer, 1-64, 1-116, 2-4
optimizing
  for code size, 2-36
  for speed, 2-36
-Os (optimize for size) compiler switch, 1-41
ostream header file, 3-34
outer loop, 2-26
output operands, 1-87
-Ov num (optimize for speed versus size) compiler switch, 1-41

P

passing
  arguments to driver, 1-48
  function parameters, 1-202
-path-install (installation location) compiler switch, 1-42
-path-output (non-temporary files location) compiler switch, 1-43
-path-temp (temporary files location) compiler switch, 1-43
-path- (tool location) compiler switch, 1-42
-pchdir (locate PCHRepository) compiler switch, 1-43
-pch (precompiled header) compiler switch, 1-43

-pedantic (ANSI standard warnings) compiler switch, 1-43
-pedantic-errors (ANSI standard errors) compiler switch, 1-44
peeled iterations, 2-61
peeling amount, 2-61
.pgo output file, 1-66
-pguide (profile-guided optimization) compiler switch, 1-44
Pipeline Viewer, 2-31
placement support keyword (section), 1-95
pm (see dual memory support keywords (pm dm))
pointer
  arithmetic action on, 1-143
  incrementing, 2-16
  induction variable, 1-117
pointer class support keyword (restrict), 1-72, 1-96
pointer-induction variables, 1-117
polar (construct from polar coordinates) functions, 4-119, 5-129
polar coordinate conversion, 4-119, 5-129
poll_flag_in (testing input flag) function, 4-121, 5-131
-P (omit line numbers and compile) compiler switch, 1-42
-P (omit line numbers) compiler switch, 1-42
power (see exp, pow, functions)
pow (power, x^y) functions, 3-177
-pplist (preprocessor listing) compiler switch, 1-44
#pragma alignment_region, 1-113
#pragma alignment_region_end, 1-113
#pragma align num, 1-112, 1-117, 2-11
#pragma all_aligned, 2-45
#pragma alloc, 1-121, 2-38
#pragma avoid_anomaly_45, 1-138
#pragma can_instantiate instance, 1-129
#pragma compiler support, 1-116
#pragma const, 1-121, 2-39
#pragma default_section, 1-133
#pragma diag, 1-135
#pragma do_not_instantiate instance, 1-128
#pragma hdrstop, 1-129
#pragma instantiate, 1-238
#pragma instantiate instance, 1-128
#pragma linkage_name, 1-132
#pragma loop_count(min, max, modulo), 1-117, 2-43
#pragma no_alias, 1-118, 2-46
#pragma no_implicit_inclusion, 1-130
#pragma no_pch, 1-131
#pragma noreturn, 1-122
#pragma no_vectorization, 1-117, 2-43
#pragma once, 1-131
#pragma optimize_as_cmd_line, 1-120, 1-137
#pragma optimize_for_space, 1-119, 1-136, 2-42
#pragma optimize_for_speed, 1-120, 1-137, 2-42
#pragma optimize_off, 1-119, 1-136, 2-42
#pragma pack (alignopt), 1-114
#pragma pad (alignopt), 1-115
#pragma pure, 1-122, 2-39
#pragma regs_clobbered, 1-123, 2-40
pragma result_alignment, 2-40
#pragma result_alignment, 1-127
#pragma retain_name, 1-132

pragmas, 1-110
alignment_region, 1-113
alignment_region_end, 1-113
align num, 1-112, 1-117
alloc, 1-121
can_instantiate instance, 1-129
code generation, 1-138
const, 1-121
data alignment, 1-111
default_section, 1-133
diag, 1-135
do_not_instantiate instance, 1-128
function side-effect, 1-120
hdrstop, 1-129
header file control, 1-129
instantiate instance, 1-128
interrupt, 1-115
interrupt handler, 1-115
linkage_name, 1-132
linking, 1-132
linking control, 1-132
loop_count(min, max, modulo), 1-117
loop optimization, 1-116, 2-43
no_alias, 1-118
no_implicit_inclusion, 1-130
no_pch, 1-131
noreturn, 1-122
no_vectorization, 1-117
once, 1-131
optimize_as_cmd_line, 1-120, 1-137
optimize_for_space, 1-119, 1-136
optimize_off, 1-119, 1-136
pack (alignopt), 1-114
pad (alignopt), 1-115
pure, 1-122
regs_clobbered string, 1-123
result_alignment, 1-127
retain_name, 1-132
section, 1-133
SIMD_for, 1-116, 1-155
system_header, 1-132
template instantiation, 1-127
vector_for, 1-118
weak_entry, 1-135
#pragma section, 1-133
#pragma SIMD_for, 1-155, 2-45
#pragma system_header, 1-132
#pragma vector_for, 1-118, 2-44
#pragma weak_entry, 1-135
#pragrma interrupt, 1-115
predefined macros, 1-166
prelinker, 1-68
preprocessing
program, 1-165
preprocessor
commands, 1-165
predefined macros, 1-166
warnings, 1-102
PrimIO device, 3-56
_primio.h header file, 3-60
__primIO label, 3-59
primiolib.c source file, 3-57
primitive I/O functions, 3-60
printable character test (see isprint function)
PRINT_CYCLES(STRING,T) macro, 3-41
printf
extending to new devices, 3-50, 3-56
printf() function, 3-56
procedural optimizations, 1-65
processor
clock rate, 3-47
time, 3-102
processor counts
measuring, 3-40
-proc (target processor) compiler switch, 1-45
profile-guided optimization, 2-6, 2-37
enabling, 1-44

profile-guided optimizations, 1-65, 1-185
program
  assignments, 1-92
  memory code storage, 1-178
  memory data storage, 1-179
  termination, 3-25
program control functions
  calloc, 3-85
  free, 3-114
  malloc, 3-166
  realloc, 3-182
-progress-rep-func compiler switch, 1-45
-progress-rep-gen-opt compiler switch, 1-45
-progress-rep-mc-opt compiler switch, 1-46
progress reporting, 1-45, 1-46
Project Options dialog box, 1-9
punctuation character test (ispunct) function, 3-145
push STS instruction, 3-22
puts macro, 1-215

Q

qsort (quicksort) function, 3-178
QUALIFIER keywords, 1-133
queue header file, 3-38

R

raise (force a signal) function, 3-180
rand function, 3-29
random number (see rand, srand functions), 3-181
rand (random number generator) function, 3-181
-R- (disable source path) compiler switch, 1-47
read function, 3-53
reads macro, 1-215
realloc (allocate used memory) function, 3-182
realloc heap function, 1-185
real matrix inversion, 4-102, 5-112
real matrix scalar multiplication, 4-111
real matrix scalar subtraction function, 4-113
__REALNAME__ macro, 1-52
real-time signals (see clear_interrupt, interruptf, interrupts, poll_flag_in, raise, signal functions)
reciprocal square root function (see rsqrt, rsqrtf function)
reciprocal square root function (see rsqrt functions)
reductions, 2-23
register names, 1-105
registers
  alternate, 1-196
  asm() constructs, 1-80
  assigning to operands, 1-80
  call preserved, 1-194
  clobbered, 1-123, 1-124
  compiler, 1-192
  performance, 1-193
  reserved, 1-47
  return, 1-126
  scratch, 1-195
  soft-wired, 1-126
  stack, 1-196
  user, 1-193
  user-reserved, 1-126
register usage (see mixed C/assembly programming)
regs_clobbered string, 1-124
remarks
  control pragma, 1-135
remove() function, 3-58
rename() function, 3-58

-reserve (reserve register) compiler switch, 1-47
reset_saturate_mode function, 1-149
reset_saturate_mode() function, 1-149
RESOLVE directive, 1-154
restore_reg macro, 1-216
restrict
  keyword, 2-30
  operator keyword, 1-96
  qualifier, 2-29
restricted pointer, 2-29
-restrict-hardware-loops compiler switch, 1-47
restrict (see pointer class support keyword (restrict))
return value transfer, 1-202
rfft_mag function, 5-133
rfftN functions, 4-123, 5-135
rframe instruction, 1-56
rms (root mean square) functions, 4-126, 5-138
-R (search for source files) compiler switch, 1-46
rsqrt (reciprocal square root) math functions, 4-127, 5-139
-rtti (enable run-time type identification) C++ mode compiler switch, 1-61
__RTTI macro, 1-61
run-time
  C/C++ environment (see mixed C/assembly programming)
  C header, 1-182
  disabling type identification, 1-60
  enabling type identification, 1-61
  header, 1-208
  header storage, 1-182
  heap section, 1-180
  heap storage, 1-180
  label, 3-184
  stack, 1-196
  stack storage, 1-179
run-time header
  using, 1-208
RUNTIME_INIT qualifier, 1-134
run-time type identification
  disable, 1-60
  enable, 1-61

S

saturated arithmetic, 1-149
saturate.h header file, 4-10, 5-13
save_reg macro, 1-216
-save-temps (save intermediate files) compiler switch, 1-48
scratch registers, 1-195
search character string (see strchr, strrchr functions)
search memory, character (see memchar function)
search path
  for include files, 1-32
  for library files, 1-34
secondary registers, 1-192, 1-196
section
  elimination, 2-36
  names, 1-177
  qualifiers, 1-133
.SECTION assembler directive, 1-95, 1-177
-section id compiler switch, 1-48
  controlling default names, 1-95
section keyword, 1-72, 1-95, 1-177
sections
  placing symbols in, 1-133
SECTKIND keywords, 1-133
SECTSTRING double-quoted string, 1-133
seek function, 3-54
seg_argv memory section, 1-185
seg_init initialization section, 1-180

segment
  keyword (see section keyword)
  legacy keyword, 1-95
segment (see Placement Support Keyword (section)
seg_rth run-time header section, 1-182
seg_stak stack section, 1-179
self-modifying code, 3-21
send string to operating system (see system function)
serial ports
  for ADSP-2116x/2126x/2136x processors, 5-13
set_alloc_type function, 1-189
set_alloc_type (set heap for dynamic memory allocation) function, 3-186
set_default_io_device, 3-57
set_flag function, 4-128, 5-140
set header file, 3-38
setjmp (define runtime label) function, 3-184
setjmp.h header file, 3-18
set jump (see longjmp, setjmp functions)
setlocale (set localization) function, 3-185
set_saturate_mode function, 1-149
set_semaphore function, 4-130, 5-142
setvbuf function, 3-26
shadow register operation, 1-158
short storage format, 1-205
-show (display command line) compiler switch, 1-48
SIGABRT handler, 3-65
signal (define signal handling) function, 3-188
signalf() function, 3-20
signalfnsm() function, 3-22
signal functions
  clear_interrupt, 3-92
  handling hardware signals, 3-18
  interrupt, 3-132
  raise, 3-180
  signal, 3-188
signal.h header file, 3-18
signals
  sig arguments, 3-92
signals (see clear_interrupt, interruptf, interrupts, poll_flag_in, raise, signal functions)
signalss() function, 3-21
-signed-bitfield (make plain bitfields signed) compiler switch, 1-50
-signed-char (make char signed) compiler switch, 1-50
__SIGNED_CHARS__ macro, 1-50, 1-171
__SIGNED_CHARS__ preprocessor macro, 1-54
__SILICON_REVISION__ macro, 1-49
SIMD_for loop
  restrictions, 1-157
SIMD_for pragma, 1-116, 1-151, 1-155, 1-160
SIMD mode, 1-150
  anomaly #40, 1-158
  avoiding anomaly #40, 1-158
  C/C++ callable subroutines, 1-222
  C/C++ constraints in, 1-157
  C/C++ data alignment rules, 1-163
  data alignment, 1-154, 1-162
  data increments, 1-160
  disabling, 1-39
  loop counter rules, 1-162
  performance in C/C++, 1-159
  processing multichannel data in, 1-151
  processing single channel data in, 1-152
  restrictions, 1-154
  SIMD_for pragma, 1-155
  using with ADSP-2116x/2126x/2136x processors, 5-17
__SIMDSHARC__ macro, 1-171

sine (see sin, sinh functions)
single case range, 1-144
sinh (sine hyperbolic) functions, 3-191
sin (sine) functions, 3-190
-si-revision (silicon revision) compiler switch, 1-49
SISD mode, 1-151
sizeof() operator, 1-98, 1-143
SKIP_SPACES macro, 1-172
slow floating-point emulation software, 1-28
sport.h header file, 4-10, 5-13
sqrt (square root) functions, 3-192
srand function, 3-29
srand (random number seed) function, 3-193
-S (stop after compilation) compiler switch, 1-47
sstream header file, 3-34
-s (strip debug information) compiler switch, 1-48
stack
  frame, 1-197
  header file, 3-38
  managing, 1-197
  managing in memory, 1-197
  managing with macros, 1-228
  pointer, 1-196
  registers, 1-196
stack frame
  ADSP-21020 processor, 1-199
  ADSP-2106x/2116x/2126x/2136x processors, 1-199
stage count, 2-65
standard
  heap management functions, 1-185
  math functions, 4-8, 5-11
  optimizations, 1-31
standard argument functions
  va_arg, 3-233
  va_end, 3-235
  va_start, 3-236
standard C library functions, 3-12 to 3-27
standard header files
  assert.h, 3-13
  ctype.h, 3-13
  errno.h, 3-14
  limits.h, 3-16
  locale.h, 3-16
  math.h, 3-16
  setjmp.h, 3-18
  signal.h, 3-18
  stdarg.h, 3-23
  stddef.h, 3-23
  stdlib.h, 3-26
  string.h, 3-27
standard I/O streams, 3-39
standard library functions
  abort, 3-65
  abs, 3-66
  acos, 3-67
  atexit, 3-73
  atoi, 3-77
  atol, 3-78
  avg, 3-82
  bsearch, 3-83
  calloc, 3-85
  clip, 3-101
  count_ones, 3-105
  div, 3-108
  exit, 3-109
  free, 3-114
  getenv, 3-116
  heap_calloc, 3-118
  heap_free, 3-120
  heap_install, 3-122
  heap_lookup_name, 3-125

heap_malloc, 3-127
heap_realloc, 3-129
labs, 3-149
lavg, 3-150
lclip, 3-151
lcount_ones, 3-152
ldiv, 3-154
lmax, 3-155
lmin, 3-156
malloc, 3-166
max, 3-167
min, 3-173
qsort, 3-178
rand, 3-181
realloc, 3-182
srand, 3-193
strtol, 3-221
strtoul, 3-223
system, 3-227
standard math functions
fmax, 4-78, 5-87
fmin, 4-79, 5-88
START_CYCLE_COUNT macro, 3-40
statement expression, 1-138
statistical
functions, 5-13
profiling, 2-5
stats.h header file, 4-10, 5-13
stdarg.h header file, 3-23
__STDC__ macro, 1-171
__STDC_VERSION__ macro, 1-171
stddef.h header file, 3-23
stderrfd function, 3-55
stdexcept header file, 3-34
stdinfd function, 3-55
stdio header file, 3-24
stdio.h header file, 3-49
stdlib.h header file, 3-26
stdoutfd function, 3-55
stdout output stream, 1-29
std references, 3-36
STOP_CYCLE_COUNT macro, 3-40
stop (see atexit, exit functions)
storage format
double, 1-27
strcat (concatenate string) function, 3-194
strchr (search character string) function, 3-195
strcmp (compare strings) function, 3-196
strcoll (compare strings, localized) function, 3-197
strcpy (copy string) function, 3-198
strcspn (compare string span) function, 3-199
stream
closing down, 3-25
streambuf class, 3-39
streambuf header file, 3-35
strerror (get error message string) function, 3-200
strftime (format a broken-down time) function, 3-201
string
literals with line breaks, 1-143
string compare (see strcmp, strcoll, strcspn, strncmp, strpbrk, strstr functions)
string concatenate (see stnrcat, strcat functions)
string conversion (see atof, atoi, atol, strtok, strtol, strxfrm functions)
string copy (see strcpy, strncpy function)
string functions
memchar, 3-168
memcmp, 3-169
memcpy, 3-170
memmove, 3-171
memset, 3-172
strcat, 3-194
strchr, 3-195
strcmp, 3-196

strcoll, 3-197
strcpy, 3-198
strcspn, 3-199
strerror, 3-200
strlen, 3-205
strncat, 3-206
strncmp, 3-207
strncpy, 3-208
strpbrk, 3-209
strrchr, 3-210
strspn, 3-211
strstr, 3-212
strtok, 3-219
strxfrm, 3-225
string header file, 3-35
string.h header file, 3-27
string length (see strlen function)
string literals
multiline, 1-36
no-multiline, 1-39
strings
converting to double, 3-213
converting to long double, 3-216
strlen (string length) function, 3-205
strncat (concatenate characters from string) function, 3-206
strncmp (compare characters in strings) function, 3-207
strncpy (copy characters in string) function, 3-208
strpbrk (compare strings, pointer break) function, 3-209
strrchr (search character string, recursive) function, 3-210
strspn (string span) function, 3-211
strstr (compare string, string) function, 3-212
strstream header file, 3-35
strtod (convert string to double) function, 3-213
strtok function, 3-29
strtok (token to string) function, 3-219
strtold (convert string to long double) function, 3-216
strtol (string to long integer) function, 3-221
strtoul (string to unsigned long tnteger) function, 3-223
-structs-do-not-overlap compiler switch, 1-51
struct tm, 3-27
strxfrm (localization transform) function, 3-225
super-fast interrupt dispatcher, 3-20
restriction with ADSP-2106x DSPs, 3-22
switches
C/C++ mode selection
-c++ (C++ mode), 1-22
C++ mode compiler switches
-no-demangle (disable demangler), 1-60
compiler common
-A (assert) compiler switch, 1-23
-aligned-stack (align stack), 1-24
-alttok (alternative tokens), 1-24
-annotate-loop-instr, 1-25
-bss (placing data in bsz), 1-25
-build-lib (build library), 1-25
-C (comments), 1-25
-c (compile only), 1-26
-compatible-pm-dm, 1-26
-const-read-write, 1-26
-D (define macro), 1-26
-debug-types, 1-26
-default-linkage-asm|-c|-c++., 1-27
-double-size-32|64, 1-27
-double-size-any, 1-28
-dryrun (terse dry-run), 1-28
-dry (verbose dry-run), 1-28

-ED (run after preprocessing to file), 1-29
-EE (run after preprocessing), 1-29
-E (stop after preprocessing), 1-29
-extra-keywords (enable short-form keywords), 1-29
-@filename (command file), 1-23
-flags (command-line input), 1-29
-float-to-int, 1-30
-force-circbuf, 1-30
-fp-associative (floating-point associative operation), 1-30
-full-version (display versions), 1-30
-g (generate debug information), 1-31
-help (command-line help), 1-32
-HH (list headers and compile), 1-31
-H (list headers), 1-31
-I directory (include search directory), 1-32
-i (less includes), 1-33
-include (include file), 1-33
-ipa (interprocedural analysis), 1-33
-I (start include directory), 1-32
-L (library search directory), 1-34
-l (link library), 1-34
-map filename (generate a memory map), 1-36
-MD (generate make rules and compile), 1-35
-mem (enable memory initialization), 1-36
-M (generate make rules only), 1-35
-MM (generate make rules and compile), 1-35
-Mo (processor output file), 1-35
-MQ (output without compilation), 1-36
-Mt filename (output make rule), 1-35
-multiline, 1-36
-no-aligned-stack (disable stack alignment), 1-36
-no-alttok (disable alternative tokens), 1-36
-no-annotate (disable alternative tokens), 1-37
-no-annotate-loop-instr, 1-37
-no-bss, 1-37
-no-builtin (no built-in functions), 1-37
-no-circbuf (no circular buffer), 1-38
-no-db (no delayed branches), 1-38
-no-defs (disable defaults), 1-38
-no-extra-keywords (disable short-form keywords), 1-38
-no-fp-associative, 1-38
-no-mem (disable memory initialization), 1-39
-no-multiline, 1-39
-no-restrict (disable restrict), 1-39
-no-saturation (no faster operations), 1-39
-no-std-ass (disable standard assertions), 1-39
-no-std-def (disable standard macro definitions), 1-40
-no-std-inc (disable standard include search), 1-40
-no-std-lib (disable standard library search), 1-40
-no-threads (disable thread-safe build), 1-40
-Oa (automatic function inlining), 1-41
-O (enable optimizations), 1-40
-Og (optimize while preserving debugging information), 1-41
-o (output file), 1-42
-Os (optimize for size), 1-41
-Ov (optimize for speed vs. size), 1-41
-path-install (installation location),

1-42
-path-output (non-temporary files location), 1-43
-path-temp (temporary files location), 1-43
-path- (tool location), 1-42
-pchdir (locate PCHRepository), 1-43
-pch (precompiled header), 1-43
-pedantic (ANSI standard warnings), 1-43
-pedantic-errors (ANSI standard errors), 1-44
-pguide (profile-guided optimization), 1-44
-P (omit line numbers), 1-42
-pplist (preprocessor listing), 1-44
-PP (omit line numbers and compile), 1-42
-proc identifier (processor), 1-45
-progress-rep-func, 1-45
-progress-rep-gen-opt, 1-45
-progress-rep-mc-opt, 1-46
-R (add source directory), 1-46
-R- (disable source path), 1-47
-reserve (reserve register), 1-47
-restrict-hardware-loops, 1-47
-save-temps (save intermediate files), 1-48
-section id, 1-48
-show (display command line), 1-48
-signed-bitfield (make plain bitfields signed), 1-50
-signed-char (make char signed), 1-50
-si-revision version (silicon revision), 1-49
sourcefile (parameter), 1-23
-S (stop after compilation), 1-47
-s (strip debug information), 1-48
-switch-pm (switch tables), 1-51
-syntax-only (system definitions), 1-52
-T filename, 1-52
-threads (enable thread-safe build), 1-52
-time (tell time), 1-53
-unsigned-bitfield (make plain bitfields unsigned), 1-53
-unsigned-char (make char unsigned), 1-54
-U (undefine macro), 1-53
-verbose, 1-54
-version (display version), 1-54
-v (version and verbose), 1-54
-warn-protos (warn if incomplete prototype), 1-56
-w (disable all warnings), 1-56
-Wdriver-limit (maximum process errors), 1-55
-Werror-limit (maximum compiler errors), 1-55
-W number (override error message), 1-55
-workaround, 1-56
-Wremarks (enable diagnostic warnings), 1-55
-write-files (enable driver I/O redirection), 1-56
-write-opts (user options), 1-57
-Wterse (enable terse warnings), 1-55
-xref (cross-reference list), 1-57
-switch-pm (switch tables) compiler switch, 1-51
symbolic names
  specifying bit definitions, 4-6, 5-8
symbols
  placing in sections, 1-133
-syntax-only (check syntax only) compiler switch, 1-51
-sysdef (system definitions) compiler switch, 1-52

sysreg_bit_* interrogation and manipulation functions, 1-106
sysreg.h header file, 1-103, 2-32, 4-10, 5-14
sysreg_read function, 1-103
sysreg_write function, 1-103
__SYSTEM__ macro, 1-52
system register bit definitions, 4-6, 5-8
system registers
  accessing from C, 1-103, 4-10, 5-14
system (send string to operating system) function, 3-227

T

tangent (see atan, atan2, cot, tan, tanh functions)
tanh (hyperbolic tangent) functions, 3-229
tan (tangent) functions, 3-228
TCOUNT register, 4-131, 4-133, 4-135, 5-144
TCOUNT registers, 4-132, 5-145
template
  class, 1-238
  function, 1-238
  instantiation pragmas, 1-127
  support in C++, 1-238
template inclusion
  control pragma, 1-130
template library header files
  algorithm, 3-37
  deque, 3-37
  functional, 3-37
  hash_map, 3-37
  hash_set, 3-37
  iterator, 3-37
  list, 3-37
  map, 3-38
  memory, 3-38
  numeric, 3-38
  queue, 3-38
  set, 3-38
  stack, 3-38
  utility, 3-38
  vector, 3-38
templates, 1-238
temporary files, 1-48
terminate (see atexit, exit functions)
testing
  specified input flag, 4-121, 5-131
thread-safe functions, 3-29
-threads (enable thread-safe build) compiler switch, 1-52
time (calendar time) function, 3-230
time.h header file, 3-27, 3-45, 3-47, 3-48
__TIME__ macro, 1-171
timer0_off function, 4-132
timer0_on function, 4-134
timer0_set function, 4-137
timer1_off function, 4-132
timer1_on function, 4-134
timer1_set function, 4-137
timer_off (disable DSP timer) function, 4-131, 5-143
timer_on (enable DSP timer) function, 4-133, 5-144
timer_set function, 4-135
timer_set (initialize DSP timer) function, 5-145
time_t data type, 3-27
-time (tell time) compiler switch, 1-53
time_t type, 3-230
time zones, 3-27
-T (linker description file) compiler switch, 1-52
tokens, string convert (see strtok function)
tolower (convert characters to lower case) function, 3-231
toupper (convert characters to upper case) function, 3-232
TPERIOD register, 4-135, 5-145

transferring
  function arguments and return value, 1-202
  function parameters to assembly routines, 1-202
trans.h header file, 4-10, 5-14
transpm (matrix transpose) functions, 4-139, 5-147
trigonometric functions (see math functions)
trip
  count, 2-65
  loop count, 2-60
  maximum, 2-65
  minimum, 2-65
  modulo, 2-65
true (see Boolean type support keywords (bool, true, false))
TST_FLAG macro, 4-128, 5-140
twidfft function, 5-149
type cast, 1-144
typeof keyword, 1-140
type sizes, data, 1-205

U

unclobbered registers, 1-125
-unsigned-bitfield (make plain bitfields unsigned) compiler switch, 1-53
-unsigned-char (make char unsigned) compiler switch, 1-54
uppercase (see isupper, toupper functions)
__USERNAME__ macro, 1-52
user registers, 1-193
utility functions
  getenv, 3-116
  system, 3-227
utility header file, 3-38
-U (undefine macro) compiler switch, 1-26, 1-53

V

va_arg (get next argument in list) function, 3-233
va_end (finish processing argument list) function, 3-235
variable
  argument macros, 1-143
  length array, 1-97, 1-142
var (variance) functions, 4-141, 5-151
va_start (initialize argument list) function, 3-236
VDK semaphores, 3-39
vecdot (vector dot product) functions, 4-142, 5-153
vecsadd (vector scalar addition) functions, 4-144, 5-155
vecsmlt (vector scalar multiplication) functions, 4-145, 5-157
vecssub (vector scalar subtraction) functions, 4-146, 5-159
vector functions, 4-11, 5-14
vector header file, 3-38
vector.h header file, 4-11, 5-14
vectorization, 2-60
  annotations, 2-62
  avoiding, 2-43
  factor, 2-61
  loop, 2-44
vecvadd (vector addition) functions, 4-147, 5-161
vecvmlt (vector multiplication) functions, 4-149, 5-163
vecvsub (vector subtraction) functions, 4-150, 5-165
-verbose (display command line) compiler switch, 1-54
-version (display version) compiler switch, 1-54
__VERSION__ macro, 1-171
__VERSIONNUM__ macro, 1-172

virtual function lookup tables, 1-48, 1-95
VisualDSP++
  debugger., 1-31
  IDDE, 1-3, 1-9
  running compiler, 1-2
  running compiler from command line, 1-5
  setting compiler options, 1-9
  simulator, 3-14, 3-24, 3-50, 3-58
voice-band compression and expansion, 5-6
volatile C program constructs, 1-87
volatile keyword, 3-49
vtbl section identifier, 1-95
-v (version and verbose) compiler switch, 1-54

W

warning messages, 1-102
warnings
  control pragma, 1-135
-Warn-protos (warn if incomplete prototype) compiler switch, 1-56
-w (disable all warnings) compiler switch, 1-56
-Wdriver-limit (maximum process errors) compiler switch, 1-55
-Werror-limit (maximum compiler errors) compiler switch, 1-55
white space character test (see isspace function)
window generators, 4-11, 5-14
window.h header file, 4-11
-W {...} number (override error message) compiler switch, 1-55
_wordsize.h header file, 3-60
workaround
  21161-anomaly-45, 1-56
  rframe, 1-56
  swfa, 1-56
-workaround compiler switch, 1-56
__WORKAROUNDS_ENABLED macro, 1-172
-Wremarks (enable diagnostic warnings) compiler switch, 1-55
-write-files (enable driver I/O pipe) compiler switch, 1-56
write function, 3-52, 3-53
-write-opts (enable driver I/O pipe) compiler switch, 1-57
writes
  array element, 2-25
writing macros, 1-172
-Wterse (enable terse warnings) compiler switch, 1-55

X

-xref (cross-reference list) compiler switch, 1-57

Z

zero
  length arrays, 1-142
  padding, 4-98, 5-136
zero_cross (count zero crossings) functions, 4-152, 5-167
ZERO_INIT qualifier, 1-134