CCES 2.9.0 C/C++ Compiler Manual for SHARC Processors

(Includes SHARC+ and ARM Processors)

Revision 2.2, May 2019

Part Number
82-100117-01

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

Copyright Information

©2019 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, Blackfin, Blackin+, CrossCore, EngineerZone, EZ-Board, EZ-KIT, EZ-KIT Lite, EZ-Extender, SHARC, SHARC+, and VisualDSP++ are registered trademarks of Analog Devices, Inc.

EZ-KIT Mini is a trademark 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 1–1
Intended Audience 1–1
Manual Contents 1–1
Technical Support 1–2
Supported Processors 1–2
Product Information 1–2
Analog Devices Website 1–3
EngineerZone 1–3
Notation Conventions 1–3

Compiler

C/C++ Compiler Overview 2–2
Compiler Components 2–3
Compiler Command-Line Interface 2–4
Running the Compiler 2–5
Processor Features 2–7
Compiler Command-Line Switches 2–9
C/C++ Mode Selection Switch Descriptions 2–18
-c89 2–18
-c99 2–18
-c++ 2–18
-c++11 2–18
-g++ 2–19
C/C++ Compiler Common Switch Descriptions 2–19
sourcefile 2–19
-@ *filename* 2–19
-A *name[tokens]* 2–19

-absolute-path-dependencies 2–20
-add-debug-libpaths 2–20
-alttok 2–20
-always-inline 2–21
-annotate 2–21
-annotate-loop-instr 2–21
-asms-safe-in-simd-for-loops 2–21
-auto-attrs 2–22
-build-lib 2–22
-C 2–22
-c 2–22
-char-size[-8|-32] 2–22
-compatible-pm-dm 2–22
-component *file.xml* 2–23
-const-read-write 2–23
-const-strings 2–23
-D *macro*[=*definition*] 2–23
-dependency-add-target *target* 2–23
-double-size[-32|-64] 2–24
-double-size-any 2–24
-dry 2–24
-dryrun 2–24
-E 2–24
-ED 2–24
-EE 2–24
-eh 2–24
-enum-is-int 2–25
-extra-keywords 2–25
-extra-precision 2–25
-file-attr *name[=value]* 2–26

-flags-{asm|compiler|ipa|lib|link|mem|prelink} *switch[, switch2[, ...]]*. 2–26
-float-to-int 2–26
-force-circbuf 2–26
-fp-associative 2–27
-full-version 2–27
-fx-contract 2–27
-fx-rounding-mode-biased 2–27
-fx-rounding-mode-truncation 2–27
-fx-rounding-mode-unbiased 2–27
-g 2–27
-glite 2–28
-gnu-style-dependencies 2–28
-H 2–28
-HH 2–28
-h[elp] 2–28
-I *directory[{,|;} directory...*] 2–28
-I- 2–29
-i 2–29
-include *filename* 2–29
-ipa 2–29
-L *directory[{;|,} directory …*] 2–30
-l *library* 2–30
-linear-simd 2–30
-list-workarounds 2–30
-loop-simd 2–30
-M 2–31
-MD 2–31
-MM 2–31
-Mo *filename* 2–31
-Mt *name* 2–31

-map *filename* 2–31
-mem 2–31
-multiline 2–31
-never-inline 2–32
-no-aligned-stack 2–32
-no-alttok 2–32
-no-annotate 2–32
-no-annotate-loop-instr 2–32
-no-assume-vols-are-iops 2–33
-no-auto-attrs 2–33
-no-circbuf 2–33
-no-const-strings 2–33
-no-db 2–33
-no-defs 2–33
-no-eh 2–34
-no-extra-keywords 2–34
-no-fp-associative 2–34
-no-fx-contract 2–34
-no-linear-simd 2–34
-no-loop-simd 2–34
-no-main-calls-exit 2–35
-no-mem 2–35
-no-multiline 2–35
-no-progress-rep-timeout 2–35
-no-rtcheck 2–35
-no-rtcheck-arr-bnd 2–35
-no-rtcheck-div-zero 2–36
-no-rtcheck-heap 2–36
-no-rtcheck-null-ptr 2–36
-no-rtcheck-shift-check 2–36
-no-rtcheck-stack 2–36
-no-rtcheck-unassigned 2–37

-no-sat-associative 2–37
-no-saturation 2–37
-no-shift-to-add 2–37
-no-simd 2–37
-no-std-ass 2–37
-no-std-def 2–38
-no-std-inc 2–38
-no-std-lib 2–38
-no-threads 2–38
-no-workaround *workaround_id[, workaround_id …]* 2–38
-normal-word-code 2–38
-nwc 2–38
-O[0|1] 2–39
-Oa 2–39
-Os 2–39
-Ov *num* 2–39
-o *filename* 2–40
-overlay 2–40
-overlay-clobbers *clobbered-regs* 2–41
-P 2–41
-PP 2–41
-p 2–41
-path-{ asm | compiler | ipa | lib | link | prelink } *pathname* 2–41
-path-install *directory* 2–42
-path-output *directory* 2–42
-path-temp *directory* 2–42
-pgo-session *session-id* 2–42
-pguide 2–42
-pplist *filename* 2–43
-proc *processor* 2–43
-prof-hw 2–43

-progress-rep-func 2–44
-progress-rep-opt 2–44
-progress-rep-timeout 2–44
-progress-rep-timeout-secs *secs* 2–44
-R *directory[{:|,} directory …]* 2–44
-R- 2–44
-reserve *register[, register …]* 2–44
-restrict-hardware-loops *maximum* 2–45
-rtcheck 2–45
-rtcheck-arr-bnd 2–45
-rtcheck-div-zero 2–46
-rtcheck-heap 2–46
-rtcheck-null-ptr 2–46
-rtcheck-shift-check 2–47
-rtcheck-stack 2–47
-rtcheck-unassigned 2–47
-S 2–47
-s 2–47
-sat-associative 2–48
-save-temps 2–48
-section*id=section_name[, id=section_name...]* 2–48
-short-word-code 2–49
-show 2–49
-si-revision *version* 2–49
-signed-bitfield 2–49
-structs-do-not-overlap 2–49
-swc 2–50
-syntax-only 2–50
-sysdefs 2–50
-T *filename* 2–50
-threads 2–50

-time 2–51
-U *macro* 2–51
-unsigned-bitfield 2–51
-v 2–52
-verbose 2–52
-version 2–52
-W{annotation|error|remark|suppress|warn} *number[, number ...]* 2–52
-Wannotations 2–52
-Werror-limit *number* 2–52
-Werror-warnings 2–52
-Wremarks 2–53
-Wterse 2–53
-w 2–53
-warn-component 2–53
-warn-protos 2–53
-workaround *workaround_id[, workaround_id…]* 2–53
-xref *filename* 2–53
C Mode (MISRA) Compiler Switch Descriptions 2–54
-misra 2–54
-misra-linkdir *directory* 2–54
-misra-no-cross-module 2–54
-misra-no-runtime 2–55
-misra-strict 2–55
-misra-suppress-advisory 2–55
-misra-testing 2–55
-Wmis_suppress *rule_number[, rule_number]* 2–55
-Wmis_warn *rule_number[, rule_number]* 2–55
MISRA-C Command-Line Switch Restrictions 2–55
C++ Mode Compiler Switch Descriptions 2–56
-anach 2–56
-check-init-order 2–57

-friend-injection 2–57
-full-dependency-inclusion 2–57
-implicit-inclusion 2–57
-no-anach 2–58
-no-friend-injection 2–58
-no-implicit-inclusion 2–58
-no-rtti 2–58
-no-std-templates 2–58
-rtti 2–58
-std-templates 2–58
Environment Variables Used by the Compiler 2–59
Data Type and Data Type Sizes 2–59
Integer Data Types 2–61
Floating-Point Data Types 2–61
Optimization Control 2–62
Optimization Levels 2–62
Interprocedural Analysis 2–64
Interaction With Libraries 2–64
Controlling Silicon Revision and Anomaly Workarounds Within the Compiler 2–65
Using the -si-revision Switch 2–65
Using the -workaround Switch 2–66
Using the -no-workaround Switch 2–66
Interactions Between the Silicon Revision and Workaround Switches 2–67
Anomalies in Assembly Sources 2–67
Using Byte-Addressing 2–67
Building Byte-Addressed Applications From C/C++ 2–68
Data Formats and Calling Conventions 2–68
Sizes of Native Types 2–68
Endianness 2–69
Symbol Names 2–70
Byte-Addressed Memory Alias 2–70

Changes to Call-Preserved Registers 2–71
Mixed Char-Size Applications 2–71
Building a Mixed Char-Size Application 2–71
How to Use Different Char Sizes Together 2–72
Type Coercions 2–73
Simple Interfaces 2–74
Constructing Complex Byte-Addressed and Word-Addressed Interfaces 2–74
The byte_addressed and word_addressed Keywords 2–74
Prohibited Constructs in Interface Regions 2–76
Using Sub-Word Types in Word-Addressed Code 2–77
Use of typedefs 2–78
Function Pointers 2–78
Using Byte-Addressed and Word-Addressed Interfaces with C++ 2–79
Assembly Code 2–79
Memory Alias Spaces Compared 2–80
Using Circular Buffers in Assembly Code 2–81
Using Word-Addressed Assembly Code Within a Byte-Addressed Application 2–82
#pragma no_stack_translation 2–82
Char-Size Agnostic Assembly Code 2–82
Defining Byte-Addressed Data in Assembly 2–83
Byte-Addressed C structs in Assembly 2–84
Using Native Fixed-Point Types 2–84
Fixed-Point Type Support 2–84
Native Fixed-Point Types 2–85
Native Fixed-Point Constants 2–86
A Motivating Example 2–86
Fixed-Point Arithmetic Semantics 2–87
Data Type Conversions and Fixed-Point Types 2–87
Bit-Pattern Conversion Functions: bitsfx and fxbits 2–88
Arithmetic Operators for Fixed-Point Types 2–89
FX_CONTRACT 2–90

Rounding Behavior 2–91
Arithmetic Library Functions 2–92
divifx 2–93
idivfx 2–93
fxdivi 2–94
mulifx 2–94
absfx 2–95
roundfx 2–95
countlsfx 2–96
strtofxfx 2–96
Fixed-Point I/O Conversion Specifiers 2–96
Setting the Rounding Mode 2–97
Language Standards Compliance 2–98
C Mode 2–98
C++ Mode 2–99
MISRA-C Compiler 2–101
MISRA-C Compiler Overview 2–101
MISRA-C Compliance 2–102
Using the Compiler to Achieve Compliance 2–102
Rules Descriptions 2–103
Run-Time Checking 2–109
Enabling Run-Time Checking 2–109
Command-Line Switches for Run-Time Checking 2–109
Pragmas for Run-Time Checking 2–110
Supported Run-Time Checks 2–110
Response When Problems Are Detected 2–111
Limitations of Run-Time Checking 2–112
C/C++ Compiler Language Extensions 2–112
Function Inlining 2–115
Inlining and Optimization 2–116
Inlining and Out-of-Line Copies 2–117

Inlining and Global asm Statements 2–117
Inlining and Sections 2–117
Inlining and Run-Time Checking 2–118
Variable Argument Macros 2–118
Restricted Pointers 2–118
Variable-Length Array Support 2–119
Non-Constant Initializer Support 2–120
Designated Initializers 2–120
Hexadecimal Floating-Point Numbers 2–122
Declarations Mixed With Code 2–122
Compound Literals 2–123
C++ Style Comments 2–123
Enumeration Constants That Are Not int Type 2–124
Boolean Type 2–124
The fract Native Fixed-Point Type 2–124
Inline Assembly Language Support Keyword (asm) 2–124
asm() Construct Syntax 2–125
asm() Construct Syntax Rules 2–126
asm() Construct Template Example 2–127
Assembly Construct Operand Description 2–127
Using long long Types in asm Constraints 2–131
Assembly Constructs With Multiple Instructions 2–132
Assembly Construct Reordering and Optimization 2–132
Assembly Constructs With Input and Output Operands 2–133
Assembly Constructs With Compile-Time Constants 2–133
Assembly Constructs and Flow Control 2–134
Guidelines on the Use of asm() Statements 2–134
Dual Memory Support Keywords (pm dm) 2–135
Memory Keywords and Assignments/Type Conversions 2–136
Memory Keywords and Function Declarations/Pointers 2–137
Memory Keywords and Function Arguments 2–138

Memory Keywords and Macros 2–138
Memory Banks 2–139
Memory Banks Versus Sections 2–139
Pragmas and Qualifiers 2–139
Memory Bank Selection 2–139
Memory Banks for Code 2–139
Memory Banks for Data 2–140
Performance Characteristics 2–141
Memory Bank Kinds 2–141
Predefined Banks 2–142
Defining Additional Banks 2–142
Placement Support Keyword (section) 2–142
Placement of Compiler-Generated Code and Data 2–143
Long Identifiers 2–143
Preprocessor Generated Warnings 2–143
Compiler Built-In Functions 2–143
builtins.h 2–144
Access to System Registers 2–144
Circular Buffer Built-In Functions 2–146
Automatic Circular Buffer Generation 2–146
Circular Buffer Increment of an Index 2–147
Circular Buffer Increment of a Pointer 2–147
Compiler Performance Built-In Functions 2–147
Expected Behavior 2–148
Known Values 2–149
Integral Built-in Functions 2–149
Floating-Point Built-in Functions 2–150
Fractional Built-In Functions 2–151
Multiplier Built-In Functions 2–153
Exclusive Transaction Built-In Functions 2–159
Miscellaneous Built-In Functions 2–160

Pragmas 2–161
Data Declaration Pragmas 2–162
#pragma align *alignopt* 2–162
#pragma alignment_region (*alignopt*) 2–163
#pragma pack (*alignopt*) 2–164
#pragma pad (*alignopt*) 2–164
#pragma no_partial_initialization 2–165
Interrupt Handler Pragmas 2–165
#pragma flush_restore_loop_stack 2–166
#pragma implicit_push_sts_handler 2–166
#pragma interrupt_complete 2–166
#pragma interrupt_complete_nesting 2–166
#pragma interrupt_dispatched_handler 2–167
#pragma interrupt_reentrant 2–167
#pragma save_restore_40_bits 2–168
#pragma save_restore_simd_40_bits 2–168
Loop Optimization Pragmas 2–168
#pragma SIMD_for 2–168
#pragma all_aligned 2–169
#pragma no_vectorization 2–169
#pragma loop_count (*min, max, modulo*) 2–169
#pragma loop_unroll *N* 2–169
#pragma no_alias 2–171
#pragma vector_for 2–171
General Optimization Pragmas 2–172
Function Side-Effect Pragmas 2–172
#pragma alloc 2–173
#pragma compatible_pm_dm_params 2–173
#pragma const 2–173
#pragma exceptret 2–173
#pragma misra_func(arg) 2–174
#pragma no_vectorization 2–174

#pragma noreturn 2–174
#pragma overlay 2–174
#pragma pgo_ignore 2–175
#pragma pure 2–175
#pragma regs_clobbered `string` 2–176
#pragma regs_clobbered_call string 2–178
#pragma result_alignment (*n*) 2–180
Class Conversion Optimization Pragmas 2–181
#pragma param_never_null `param_name [ ... ]` 2–181
#pragma suppress_null_check 2–182
Template Instantiation Pragmas 2–183
#pragma instantiate *instance* 2–183
#pragma do_not_instantiate *instance* 2–184
#pragma can_instantiate *instance* 2–184
Header File Control Pragmas 2–184
#pragma no_implicit_inclusion 2–184
#pragma once 2–185
#pragma system_header 2–185
Fixed-Point Arithmetic Pragmas 2–185
#pragma FX_CONTRACT {ON|OFF} 2–185
#pragma FX_ROUNDING_MODE {TRUNCATION|BIASED|UNBIASED} 2–186
#pragma STDC FX_FULL_PRECISION {ON|OFF|DEFAULT} 2–186
#pragma STDC FX_FRACT_OVERFLOW {SAT|DEFAULT} 2–186
Inline Control Pragmas 2–186
#pragma always_inline 2–187
#pragma inline 2–187
#pragma never_inline 2–187
Linking Control Pragmas 2–188
#pragma linkage_name *identifier* 2–188
#pragma function_name *identifier* 2–188
#pragma core 2–188
#pragma retain_name 2–191

#pragma section/#pragma default_section 2–192
#pragma file_attr("name[=value]"[, "name[=value]" [...]]) 2–195
#pragma weak_entry 2–195
Diagnostic Control Pragmas 2–196
Modifying the Severity of Specific Diagnostics 2–196
Modifying the Behavior of an Entire Class of Diagnostics 2–197
Saving or Restoring the Current Behavior of All Diagnostics 2–197
Run-Time Checking Pragmas 2–198
#pragma rtcheck(off) 2–198
#pragma rtcheck(on) 2–198
Memory Bank Pragmas 2–198
#pragma code_bank(*bankname*) 2–199
#pragma data_bank(*bankname*) 2–199
#pragma stack_bank(*bankname*) 2–200
#pragma default_code_bank(*bankname*) 2–200
#pragma default_data_bank(*bankname*) 2–201
#pragma default_stack_bank(*bankname*) 2–201
#pragma bank_memory_kind(*bankname, kind*) 2–201
#pragma bank_read_cycles(*bankname, cycles[, bits]*) 2–201
#pragma bank_write_cycles(*bankname, cycles[, bits]*) 2–202
#pragma bank_maximum_width(*bankname, width*) 2–202
Code Generation Pragmas 2–202
#pragma avoid_anomaly_45 {on | off} 2–203
#pragma no_db_return 2–203
Exceptions Table Pragma 2–203
#pragma generate_exceptions_tables 2–204
Mixed Char-Size Interface Pragmas 2–204
#pragma byte_addressed [(push)|(pop)] 2–205
#pragma word_addressed [(push)|(pop)] 2–205
#pragma default_addressed [(push)|(pop)] 2–205
#pragma no_stack_translation 2–205
GCC Compatibility Extensions 2–206

Statement Expressions 2–206
Type Reference Support Keyword (Typeof) 2–207
Generalized Lvalues 2–208
Conditional Expressions With Missing Operands 2–208
Zero-Length Arrays 2–208
GCC Variable Argument Macros 2–208
Line Breaks in String Literals 2–209
Arithmetic on Pointers to Void and Pointers to Functions 2–209
Cast to Union 2–209
Ranges in Case Labels 2–209
Escape Character Constant 2–209
Alignment Inquiry Keyword (__alignof__) 2–209
Keyword for Specifying Names in Generated Assembler (asm) 2–209
Function, Variable and Type Attribute Keyword (__attribute__) 2–210
Unnamed struct/union Fields Within struct/unions 2–211
Support for 40-Bit Arithmetic 2–212
Using 40-Bit Arithmetic in Compiled Code 2–212
Run-Time Library Functions That Use 40-Bit Arithmetic 2–212
SIMD Support 2–213
A Brief Introduction to SIMD Mode 2–214
What the Compiler Can Do Automatically 2–214
What Prevents the Compiler From Automatically Exploiting SIMD Mode 2–214
How to Help the Compiler Exploit SIMD Mode 2–215
How to Prevent SIMD Code Generation 2–216
Accessing External Memory on ADSP-2126x and ADSP-2136x Processors 2–217
Link-Time Checking of Data Placement 2–217
Inline Functions for External Memory Access 2–217
Preprocessor Features 2–217
Predefined Preprocessor Macros 2–217
Writing Macros 2–229
Compound Macros 2–229

C/C++ Run-Time Model and Environment 2–231
Registers 2–231
Dedicated Registers 2–233
Mode Registers 2–234
Preserved Registers 2–234
Scratch Registers 2–235
Stack Registers 2–236
Parameter Registers 2–236
Return Registers 2–236
Aggregate Return Register 2–236
Reservable Registers 2–236
Alternate Registers 2–237
Managing the Stack 2–237
Function Call and Return 2–243
Transferring Function Arguments and Return Value 2–244
Basic Argument Passing 2–244
Passing Parameters for Variable Argument Lists 2–245
Passing a C++ Class Instance 2–246
Return Values 2–246
Parameter and Return Value Examples 2–246
Calling Assembly Subroutines From C/C++ Programs 2–248
Calling C/C++ Functions From Assembly Programs 2–248
C/C++/Assembly Support Macros 2–248
Symbol Names in C/C++ and Assembly 2–250
C/C++ and Assembly: Extern Linkage 2–251
C and Assembly: Underscore Prefix or Dot Suffix 2–251
Other Approaches 2–251
Implementing C++ Member Functions in Assembly Language 2–252
Writing C/C++-Callable SIMD Subroutines 2–253
Mixed C/C++/Assembly Programming Examples 2–253
Using Inline Assembly 2–254
Using Macros to Manage the Stack 2–255

Stack Alignment 2–256
Using Scratch Registers 2–256
Using Void Functions 2–257
Using the Stack for Arguments 2–258
Using Registers for Arguments and Return 2–259
Using Non-Leaf Routines That Make Calls 2–259
Using Call Preserved Registers 2–261
Exceptions Tables in Assembly Routines 2–262
Data Storage Formats 2–266
Using Data Storage Formats 2–266
Floating-Point Data Size 2–267
Floating-Point Binary Formats 2–269
IEEE Floating-Point Format 2–269
IEEE Floating-Point Implementation 2–270
fract Data Representation 2–270
Precision Restrictions With 40-Bit Floating-Point Arithmetic 2–271
Memory Section Usage 2–272
Code Storage in Program Memory 2–274
Data Storage in Data Memory 2–274
Data Storage in Program Memory 2–274
Run-Time Stack Storage 2–274
Run-Time Heap Storage 2–275
Initialization Data Storage 2–275
Global Array Alignment 2–275
Controlling System Heap Size and Placement 2–276
Managing the System Heap in the IDE 2–276
Managing the System Heap in the .LDF File 2–276
Standard Heap Interface 2–278
Using Multiple Heaps 2–278
Defining a Heap 2–279
Defining Additional Heaps in the IDE 2–279
Defining Heaps at Runtime 2–280

Tips for Working With Heaps 2–280
Allocating C++ STL Objects to a Non-Default Heap 2–280
Using the Alternate Heap Interface 2–282
C++ Run-Time Support for the Alternate Heap Interface 2–283
Freeing Space 2–283
Startup and Termination 2–284
Memory Initialization 2–284
Bootable Images 2–284
Non-Bootable Images 2–285
Global Constructors 2–285
Constructors and Destructors of Global Class Instances 2–286
Constructors, Destructors and Memory Placement 2–287
Support for argv/argc 2–287
Compiler C++ Template Support 2–287
Template Instantiation 2–288
Exported Templates 2–288
Implicit Instantiation 2–288
Generated Template Files 2–289
Identifying Un-Instantiated Templates 2–289
File Attributes 2–291
Automatically-Applied Attributes 2–291
Content Attributes 2–292
FuncName Attributes 2–292
Encoding Attributes 2–292
Default LDF Placement 2–293
Sections Versus Attributes 2–293
Granularity 2–293
"Hard" Versus "Soft" 2–294
Number of Values 2–294
Using Attributes 2–294
Example 1 2–294

Example 2 2–296
Implementation Defined Behavior 2–296
Enumeration Type Implementation Details 2–296
ISO/IEC 9899:1990 C Standard (C89 Mode) 2–297
G3.1 Translation 2–297
G3.2 Environment 2–297
G3.3 Identifiers 2–297
G3.4 Characters 2–297
G3.5 Integers 2–298
G3.6 Floating-Point 2–300
G3.7 Arrays and Pointers 2–301
G3.8 Registers 2–301
G3.9 Structures, Unions, Enumerations and Bit-Fields 2–301
G3.10 Qualifiers 2–302
G3.11 Declarators 2–302
G3.12 Statements 2–302
G3.13 Preprocessing Directives 2–302
G3.14 Library Functions 2–303
ISO/IEC 9899:1999 C Standard (C99 Mode) 2–306
J3.1 Translation 2–306
J3.2 Environment 2–307
J3.3 Identifiers 2–308
J3.4 Characters 2–308
J3.5 Integers 2–310
J3.6 Floating-Point 2–311
ISO/IEC 14822:2003 C++ Standard (C++ Mode) 2–312
1.7 The C++ Memory Model 2–312
1.9 Program Execution 2–312
2.1 Phases of Translation 2–312
2.2 Character Sets 2–312
2.13.2 Character Literals 2–313
2.13.4 String Literals 2–313

3.6.1 Main Function 2–313
3.6.2 Initialization of Non-Local Objects 2–314
3.9 Types 2–314
3.9.1 Fundamental Types 2–314
3.9.2 Compound Types 2–315
4.7 Integral Conversions 2–315
4.8 Floating-Point Conversions 2–315
4.9 Floating-Integral Conversions 2–315
5.2.8 Type Identification 2–315
5.2.10 Reinterpret Cast 2–315
5.3.3 Sizeof 2–316
5.6 Multiplicative Operators 2–316
5.7 Additive Operators 2–317
5.8 Shift Operators 2–317
7.1.5.2 Simply Type Specifiers 2–317
7.2 Enumeration Declarations 2–317
7.4 The asm Declaration 2–317
7.5 Linkage Specifications 2–317
9.6 Bit-Fields 2–318
14 Templates 2–318
14.7.1 Implicit Instantiation 2–318
15.5.1 The terminate() Function 2–319
15.5.2 The unexpected() Function 2–319
16.1 Conditional Inclusion 2–319
16.2 Source File Inclusion 2–319
16.6 Pragma Directive 2–320
16.8 Predefined Macro Names 2–320
17.4.4.5 Reentrancy 2–320
17.4.4.8 Restrictions on Exception Handling 2–320
18.3 Start and Termination 2–321
18.4.2.1 Class bad_alloc 2–321
18.5.1 Class type_info 2–321

18.5.2 Class bad_cast 2–322
18.5.3 Class bad_typeid 2–322
18.6.1 Class Exception 2–322
18.6.2.1 Class bad_exception 2–322
21 Strings Library........ 2–322
21.1.3.2 struct char_traits<wchar_t> 2–323
22.1.1.3 Locale Members 2–323
22.2.1.3 ctype Specializations 2–323
22.2.1.3.2 ctype<char> Members 2–323
22.2.5.1.2 time_get Virtual Functions 2–323
22.2.5.3.2 time_put Virtual Functions 2–324
22.2.7.1.2 Messages Virtual Functions 2–324
26.2.8 Complex Transcendentals 2–325
27.1.2 Positioning Type Limitations 2–325
27.4.1 Types 2–325
27.4.2.4 ios_base Static Members 2–325
27.4.4.3 basic_ios iostate Flags Functions 2–325
27.7.1.3 Overridden Virtual Functions 2–325
27.8.1.4 Overridden Virtual Functions 2–325
C.2.2.3 Macro NULL 2–326
D.6 Old iostreams Members 2–326

Optimal Performance from C/C++ Source Code

General Guidelines 3–2
How the Compiler Can Help 3–2
Using the Compiler Optimizer 3–3
Using Compiler Diagnostics 3–3
Warnings, Annotations and Remarks 3–3
Run-Time Diagnostics 3–4
Steps for Developing Your Application 3–4
Using Profile-Guided Optimization 3–5
Using Profile-Guided Optimization With a Simulator 3–6

Using Profile-Guided Optimization With Hardware 3–7
Profile-Guided Optimization and Multiple Source Uses 3–10
Profile-Guided Optimization and the `-Ov num` Switch 3–11
Profile-Guided Optimization and Multiple PGO Data Sets 3–11
When to Use Profile-Guided Optimization 3–11
Using Interprocedural Optimization 3–11
The volatile Type Qualifier 3–12
Data Types 3–12
Avoiding Emulated Arithmetic 3–13
Getting the Most From IPA 3–14
Initialize Constants Statically 3–14
Dual Word-Aligning Your Data 3–15
Using the aligned() Builtin 3–15
Avoiding Aliases 3–16
Indexed Arrays Versus Pointers 3–18
Trying Pointer and Indexed Styles 3–18
Using Function Inlining 3–18
Using Inline asm Statements 3–19
Memory Usage 3–20
Improving Conditional Code 3–20
Using PGO in Function Profiling 3–23
Example of Using Profile-Guided Optimization 3–23
Opening the Project 3–23
Gathering the Profile 3–23
Rebuilding With the Profile 3–24
Loop Guidelines 3–25
Keeping Loops Short 3–25
Avoiding Unrolling Loops 3–25
Avoiding Loop Rotation by Hand 3–26
Avoiding Complex Array Indexing 3–26
Inner Loops Versus Outer Loops 3–27

Avoiding Conditional Code in Loops 3–27
Avoiding Placing Function Calls in Loops 3–28
Avoiding Non-Unit Strides 3–28
Loop Control 3–28
Using the Restrict Qualifier 3–29
Using Built-In Functions in Code Optimization 3–30
Using System Support Built-In Functions 3–30
Using Circular Buffers 3–31
Smaller Applications: Optimizing for Code Size 3–32
Using Pragmas for Optimization 3–33
Function Pragmas 3–33
#pragma alloc 3–33
#pragma const 3–34
#pragma pure 3–34
#pragma result_alignment 3–34
#pragma regs_clobbered 3–35
#pragma optimize_{off|for_speed|for_space|as_cmd_line} 3–36
Loop Optimization Pragmas 3–36
#pragma loop_count 3–36
#pragma no_vectorization 3–37
#pragma vector_for 3–37
#pragma SIMD_for 3–37
#pragma all_aligned 3–37
#pragma no_alias 3–38
Useful Optimization Switches 3–38
How Loop Optimization Works 3–39
Terminology 3–39
Loop Optimization Concepts 3–41
Software Pipelining 3–42
Loop Rotation 3–42
Loop Vectorization 3–44

Modulo Scheduling ... 3–45
Initiation Interval (II) and the Kernel ... 3–45
Minimum Initiation Interval Due to Resources (Res MII) ... 3–47
Minimum Initiation Interval Due to Recurrences (Rec MII) ... 3–48
Stage Count (SC) ... 3–48
Variable Expansion and MVE Unroll ... 3–49
Trip Count ... 3–53
A Worked Example ... 3–53
Assembly Optimizer Annotations ... 3–55
Annotation Examples ... 3–56
Importing Annotation Examples ... 3–57
Viewing Annotation Examples in the IDE ... 3–57
Viewing Annotation Examples in Generated Assembly ... 3–58
Global Information ... 3–58
Procedure Statistics ... 3–59
Instruction Annotations ... 3–60
Loop Identification ... 3–60
Loop Identification Annotations ... 3–61
File Position ... 3–62
Vectorization ... 3–63
Unroll and Jam ... 3–63
Loop Flattening ... 3–64
Vectorization Annotations ... 3–65
Modulo Scheduling Information ... 3–65
Annotations for Modulo Scheduled Instructions ... 3–66
Warnings, Failure Messages and Advice ... 3–69
Analyzing Your Application ... 3–72
Application Analysis Configuration ... 3–72
Application Analysis and File Naming ... 3–72
Device for Profiling Output ... 3–73
Frequency of Flushing Profile Data ... 3–73

Profiling With Instrumented Code 3–74
Generating an Application With Instrumented Profiling 3–74
Running the Executable File 3–74
Generating an Instrumented Profiling Report 3–75
Invoking the `instrprof.exe` Command-Line Reporter 3–75
Contents of the Profiling Report 3–76
Reporter Tool Report Format 3–77
The instrprof Command Line Tool Report Format 3–77
Profiling Data Storage 3–78
Computing Cycle Counts 3–78
Multi-Threaded and Non-Terminating Applications 3–78
Flushing Profile Data 3–78
Profiling of Interrupts and Kernel Time 3–79
Behavior That Interferes With Instrumented Profiling 3–79
Profile-Guided Optimization and Code Coverage 3–80
Heap Debugging 3–80
Getting Started With Heap Debugging 3–82
Linking With the Heap Debugging Library 3–82
Heap Debugging Macro 3–82
Default Behavior 3–83
Additional Heap Overheads 3–83
Heap Debugging Report 3–83
Using the Heap Debugging Library 3–84
Detected Errors 3–84
Viewing Reports 3–85
stderr Diagnostics 3–86
Call Stack 3–87
Setting the Severity of Error Messages 3–87
Default Diagnostic Severities 3–89
Guard Regions 3–89
Enabling and Disabling Features 3–91
Buffering 3–92

Pausing Heap Debugging 3–93
Finishing Heap Debugging 3–93
Verifying Heaps 3–94
Behavior of Heap Debugging Library 3–94
Unfreed File I/O Buffers 3–95
Memory Used by Operating Systems 3–95
Generating a Heap Debugging Report 3–95
Heap Debugging And -char-size-32 3–96
Stack Overflow Detection 3–96
About Stack Overflows 3–96
What is Stack Overflow? 3–97
Likely Causes of Stack Overflow 3–97
Difficulties in Diagnosing Stack Overflow 3–97
Stack Overflow Detection Facility 3–98
Limitations on the Compiler's Stack Detection Capability 3–98
Fixing a Stack Overflow 3–98
Reporter.exe Command Line Report Generation Utility 3–98

1 Preface

Thank you for purchasing CrossCore® Embedded Studio (CCES), Analog Devices development software for SHARC®and SHARC+® digital-signal processors.

Purpose of This Manual

The *C/C++ Compiler Manual for SHARC Processors* contains information about the C/C++ compiler for SHARC (ADSP-211xx, ADSP-212xx, ADSP-213xx and ADSP-214xx) and SHARC+ (ADSP-215xx and ADSP-SCxxx) 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 are programmers who are familiar with Analog Devices SHARC processors. This manual assumes that the audience has a working knowledge of the SHARC processors' architecture and instruction set and C/C++ programming languages.

Programmers who are unfamiliar with SHARC processors can use this manual, but should supplement it with other texts (such as the appropriate hardware reference, programming reference, and data sheet) that provide information about their SHARC processor architecture and instructions.

Manual Contents

The manual consists of:

- Chapter 1, Compiler

  Provides information on compiler options, language extensions, and C/C++/assembly interfacing.

- Chapter 2, Optimal Performance from C/C++ Source Code

  Shows how to optimize compiler operation.

Technical Support

You can reach Analog Devices processors and DSP technical support in the following ways:

- Post your questions in the processors and DSP support community at EngineerZone®:

  http://ez.analog.com/community/dsp

- Submit your questions to technical support directly at:

  http://www.analog.com/support

- E-mail your questions about processors, DSPs, and tools development software from ***CrossCore Embedded Studio*** or ***VisualDSP++***®:

  Choose ***Help*** > ***Email Support***. This creates an e-mail to processor.tools.support@analog.com and automatically attaches your CrossCore Embedded Studio or VisualDSP++ version information and `license.dat` file.

- E-mail your questions about processors and processor applications to:

  processor.tools.support@analog.com

  processor.china@analog.com

- Contact your Analog Devices sales office or authorized distributor. Locate one at:

  http://www.analog.com/adi-sales

- Send questions by mail to:

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

Supported Processors

The names *SHARC* and *SHARC+* refer 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. Refer to the CCES online help for a complete list of supported processors.

Product Information

Product information can be obtained from the Analog Devices website and CrossCore Embedded Studio online help system.

Analog Devices Website

The Analog Devices website, http://www.analog.com, provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

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

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

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

EngineerZone

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

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

Notation Conventions

Text conventions used in this manual are identified and described as follows. Additional conventions, which apply only to specific chapters, can appear throughout this document.

Example	*Description*
File > Close	Titles in bold style indicate the location of an item within the CrossCore Embedded Studio IDE'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 ellipsis; 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 `letter gothic` font and italic style format.

Example	*Description*
NOTE:	*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:	*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.
ATTENTION:	*ATTENTION:* 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 *ATTENTION:* appears instead of this symbol.

2 Compiler

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

This chapter contains:

- C/C++ Compiler Overview provides an overview of C/C++ compiler for SHARC processors.
- Compiler Command-Line Interface describes the compiler operation as it processes programs, including input and output files, and command-line switches.
- Using Byte-Addressing describes what byte-addressing means, how to build a byte-addressed application, and how to use existing word-addressed code in a new byte-addressed application.
- Using Native Fixed-Point Types describes the compiler support for the native fixed-point type fract. The fract data type is defined in Chapter 4 of the *Extensions to support embedded processors* ISO/IEC draft technical report TR 18037.
- Language Standards Compliance describes how to enable the compliance to the ISO/IEC 9899:1990 C standard, ISO/IEC 9899:1999 C standard, ISO/IEC 14882:2003 C++ standard, or ISO/IEC 14882:2011 C++ standard.
- MISRA-C Compiler describes how the cc21k compiler enables checking for MISRA-C Guidelines.
- Run-Time Checking describes the additional run-time checks supported by the compiler.
- C/C++ Compiler Language Extensions describes the cc21k compiler extensions to the ISO/ANSI standard for the C and C++ languages.
- Preprocessor Features contains information on the preprocessor and ways to modify source compilation.
- C/C++ Run-Time Model and Environment contains reference information about implementation of C/C++ programs, data, and function calls in the ADSP-21xxx and ADSP-SCxxx processors.
- Compiler C++ Template Support describes how templates are instantiated at compilation time.
- File Attributes describes how file attributes help with the placement of run-time library functions.
- Implementation Defined Behavior describes how the compiler implements various language features where the standard allows flexibility.

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 processor memory.

Using C/C++, developers can significantly decrease time-to-market since it gives them the ability to work efficiently 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 compiles ISO/ANSI standard C and C++ code for SHARC processors. Additionally, Analog Devices includes, within the compiler, a number of C language extensions designed to aid processor development. The compiler runs from the CCES environment or from an operating system command line.

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

The `cc21k` compiler processes your source files containing the C/C++ program. The compiler supports the following standards, each with Analog Devices extensions enabled:

- A freestanding implementation of the ISO/IEC 9899:1990 C standard (C89).
- A freestanding implementation of the ISO/IEC 9899:1999 C standard (C99).
- A freestanding implementation of the ISO/IEC 14882:2003 C++ standard (C++ 2003). The compiler supports the language features supported by a standard subset of the C++ Library. You can obtain C++ library (*Dinkum EC++ Library*) reference text in the CCES online help.
- A freestanding implementation of the ISO/IEC 14882:2011 C++ standard (C++ 2011).

RTTI and exceptions for C++ are supported, but disabled by default. See the `-rtti` and `-eh` the switches for more information.

For information on the C or C++ language standards, see any of the many reference texts.

The `cc21k` compiler supports a set of C/C++ language extensions. These extensions support hardware features of the ADSP-21xxx and ADSP-SCxxx processors. For information on these extensions, see C/C++ Compiler Language Extensions.

You can set the compiler options for the project in the ***Preferences*** dialog box of the CCES Integrated Development Environment (IDE). These options control how the compiler processes the source files, letting you select features that include the language dialect, error reporting, and debugger output.

Access the ***Preferences*** pages from the ***Project*** > ***Properties*** menu. Within the ***Preferences*** pages, navigate to ***C/C++ Build*** > ***Settings***. Alternatively, click the ***Settings*** icon in the ***Project Explorer*** view. For both routines, access the compiler options via ***Settings*** > ***Tool Settings*** > ***CrossCore SHARC C/C++ Compiler***.

For more information on the CCES environment, see the CCES online help.

Compiler Components

The compiler is not a single program, but a collection of programs, each with a different task:

Compiler Driver

The compiler driver, `cc21k`, is the user interface to other programs, and is the program to invoke when you run the compiler on the command line. Its responsibility is to marshall and interpret the command-line arguments to determine what other components and code-generation tools to invoke, and in what order. The compiler driver hides the complexity and presents a consistent interface. For this reason, throughout the documentation, "the compiler", "compiler driver", and "`cc21k`" are used interchangeably.

Compiler Proper

The compiler proper, found in `..\SHARC\etc\compiler` and/or `..\SHARC\etc\compiler_ba`, is the actual compiler. It compiles a single C/C++ source file into a single assembly output file. The compiler driver invokes the compiler proper for each C/C++ source file specified.

Assembler

The assembler, `easm21k`, assembles a single assembly source file into a single object file. The compiler driver invokes the assembler to translate both user-supplied assembly files and compiler-generated assembly files.

Linker

The linker, `linker`, combines object files into executable files, and searches library files to resolve references to undefined symbols. The linker relies on an `.ldf` file to specify how the resulting collection of symbols are mapped into memory. The compiler driver invokes the linker when the specified output file is an executable file.

Prelinker

The prelinker is found in `..\SHARC\etc\prelinker`. Its purpose is to examine the set of objects and libraries before linking. The prelinker also instructs the compiler driver to recompile files or add other libraries or switches, as needed. The compiler driver invokes the prelinker just before invoking the linker. Language features supported by the prelinker include:

- C++ template instantiation
- Interprocedural analysis

- Instrumented profiling

IPA Solver

The IPA Solver, found in `..\SHARC\etc\ipa`, propagates information between compiled modules, as part of Interprocedural Analysis. If propagated information can improve optimization, the IPA Solver directs the compiler driver to recompile a source file. The prelinker invokes the IPA Solver when any of the input files are compiled with IPA optimization enabled.

PGO Merger

The PGO merger, found in `..\SHARC\etc\pgo`, combines multiple profiles gathered through profiled executions of an application. The PGO merger also produces a single profile for the compiler to use. The compiler driver invokes the PGO merger whenever more than one PGO profile is specified.

Librarian

The librarian, `elfar`, provides facilities for creating, modifying, and inspecting library files. The compiler driver invokes the librarian when the output file is a library file.

Memory Initializer

The memory initializer, `meminit`, creates an initialization stream within the executable file. The compiler driver directs the linker to invoke the memory initializer after linking, when the `-mem` switch is specified.

The assembler, linker, and librarian are documented in the *Assembler and Preprocessor Manual* and *Linker and Utilities Manual*. Invoke the other components only through the compiler driver, never directly.

Compiler Command-Line Interface

This section describes how the `cc21k` compiler is invoked from the command line and the various types of files used by and generated from the compiler. It also describes the switches used to tailor the compiler operation.

This section contains:

- Running the Compiler
- Compiler Command-Line Switches
- Environment Variables Used by the Compiler
- Data Type and Data Type Sizes
- Optimization Control
- Controlling Silicon Revision and Anomaly Workarounds Within the Compiler

By default, the compiler runs with Analog Devices extension keywords for C code enabled. It means that the compiler processes source files written in ISO/IEC 9899:1999 standard C language, supplemented with Analog Devices extensions. The ***File Extensions Specifying Compiler Action*** table (in Running the Compiler) 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 the ***Input and Output File Extensions*** table (in Running the Compiler).

The ***C/C++ Mode Selection Switches*** table (in Compiler Command-Line Switches) 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 the ***C/C++ Compiler Common Switches*** table (in Compiler Command-Line Switches). A summary of the C++-specific compiler switches appears in the ***C++ Mode Compiler Switches*** table (in Compiler Command-Line Switches). Each switch description follows the summaries.

NOTE: When developing a project, sometimes it is useful to modify the default option settings of the compiler. The way the compiler options are set depends on the environment used to run the processor development software. For more information, see Environment Variables Used by the Compiler.

Running the Compiler

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

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

The ***cc21k Command-Line Syntax*** table describes the syntax elements.

Table 2-1: cc21k Command-Line Syntax

Command Element	*Description*
`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 *`sourcefile`* element 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.

NOTE: When file names or switches for the compiler include spaces or special characters, ensure that the names are properly quoted, usually using double-quote characters. The properly quoted file names ensure that the operating system does not interpret the names before passing them to the compiler.

The `cc21k` compiler uses the file extension to determine what the file contains and what operations to perform upon it. The ***Input and Output File Extensions*** table lists the allowed extensions.

Table 2-2: Input and Output File Extensions

File Extension	*File Extension Description*
`.c .C`	C source file
`.cpp .cxx .cc .c++`	C++ source file
`.h`	Header file (referenced by an `#include` statement)
`.hpp .hh .hxx .h++`	C++ header file (referenced by a `#include` statement)
`.hpl`	Heap debugging output file-used by the Reporter Tool to produce a report on heap usage and related errors
`.ii .ti`	Template instantiation files-used internally by the compiler when instantiating templates
`.et`	Exported template files-used internally by the compiler when instantiating exported templates
`.ipa`	Interprocedural analysis files-used internally by the compiler when performing interprocedural analysis.
`.pgo .pgi .pgt`	Execution profile generated by a simulation run or instrumented executable
`.i`	Preprocessed source file-created when preprocess only is specified
`.s .asm`	Assembly language source files
`.is`	Preprocessed assembly language source-retained when `-save-temps` is specified.
`.sbn`	Binary data included by an assembly language source file
`.ldf`	Linker description file
`.misra`	Text file used by prelinker for MISRA-C Guidelines checking
`.doj .o`	Object file to be linked
`.dlb .a`	Library of object files to be linked as needed
`.dxe`	Executable file produced by compiler
`.xml`	Processor memory map file output
`.sym`	Processor symbol map file output

The 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 file names and by various switches.

In normal operation, the compiler uses the files listed in the ***File Extensions Specifying Compiler Action*** table to perform a specified action.

Table 2-3: File Extensions Specifying Compiler Action

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

Table 2-3: File Extensions Specifying Compiler Action (Continued)

Extension	*Action*
`.pgo .pgi`	Profile-guided optimization information file is used during compilation.

If multiple files are specified, each is processed to produce an object file and then all the object files are presented to the linker.

You can stop this sequence at various points using appropriate compiler switches (`-E`, `-P`, `-M`, `-H`, `-S`, and `-c`), or by selecting options within the IDE.

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. The ***Input and Output File Extensions*** table 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. The programs search directories that you specify and path information that you include in the file name. The ***Input and Output File Extensions*** table indicates the extensions that the preprocessor, compiler, assembler, and linker support. The compiler supports relative and absolute directory names to define file extension paths. For information on additional search directories, see the command-line switch that controls the specific type of extensions.

When providing an input or output file name as an optional parameter, follow these guidelines.

- Use a file name (include the file extension) with an unambiguous relative path or an absolute path. A file name with an absolute path includes the directory, file name, and file extension. The compiler uses the file extension convention listed in the ***Input and Output File Extensions*** table 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.

NOTE: Use the verbose output switches for the preprocessor, compiler, assembler, and linker to cause each of these tools to display command-line information as they process each file.

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 for more information.

Processor Features

Some processor features are only supported by a subset of the ADSP-21xxx and ADSP-SCxxx processors. These features are VISA (Variable Instruction Set Architecture) execution, byte-addressing, and native double-precision floating-point arithmetic.

The ***Processors Supporting VISA Execution*** table shows the processors that support VISA execution. The compiler targets the VISA instruction set when using the `-short-word-code` switch, discussed in `-short-word-code`.

Table 2-4: Processors Supporting VISA Execution

ADSP-214xx Processors	*ADSP-215xx Processors*	*ADSP-SC5xx Processors*
ADSP-21467 ADSP-21469	ADSP-21562 ADSP-21563 ADSP-21565 ADSP-21566 ADSP-21567 ADSP-21569	ADSP-SC570 ADSP-SC571 ADSP-SC572 ADSP-SC573
ADSP-21477 ADSP-21479	ADSP-21571 ADSP-21573	ADSP-SC582 ADSP-SC583 ADSP-SC584 ADSP-SC587 ADSP-SC589
ADSP-21483 ADSP-21486 ADSP-21487 ADSP-21488 ADSP-21489	ADSP-21583 ADSP-21584 ADSP-21587	

The processors that support byte-addressing are given in the ***Processors Supporting Byte-Addressing*** table. For more information on byte-addressing, see Using Byte-Addressing.

Table 2-5: Processors Supporting Byte-Addressing

All ADSP-215xx Processors	*All ADSP-SC5xx Processors*
ADSP-21562 ADSP-21563 ADSP-21565 ADSP-21566 ADSP-21567 ADSP-21569	ADSP-SC570 ADSP-SC571 ADSP-SC572 ADSP-SC573
ADSP-21571 ADSP-21573	ADSP-SC582 ADSP-SC583 ADSP-SC584 ADSP-SC587 ADSP-SC589

The processors that support native double-precision floating-point arithmetic are given in the ***Processors Supporting Native Double-Precision Floating-Point Arithmetic*** table. For more information on the performance implications of

native vs. emulated arithmetic, see Avoiding Emulated Arithmetic in the Optimal Performance from C/C++ Source Code chapter.

Table 2-6: Processors Supporting Native Double-Precision Floating-Point Arithmetic

All ADSP-215xx Processors	*All ADSP-SC5xx Processors*
ADSP-21562 ADSP-21563 ADSP-21565 ADSP-21566 ADSP-21567 ADSP-21569	ADSP-SC570 ADSP-SC571 ADSP-SC572 ADSP-SC573
ADSP-21571 ADSP-21573	ADSP-SC582 ADSP-SC583 ADSP-SC584 ADSP-SC587 ADSP-SC589
ADSP-21583 ADSP-21584 ADSP-21587	

Compiler Command-Line Switches

This section describes the command-line switches (options) used when compiling. It contains a set of tables providing a brief description of each switch. The tables are organized by type of switch. Following the tables are sections providing fuller descriptions of each switch.

These tables summarize generic and specific switches.

- ***C/C++ Mode Selection Switches***
- ***C/C++ Compiler Common Switches***
- ***C Mode (MISRA) Compiler Switches***
- ***C++ Mode Compiler Switches***

A brief description of each switch follows the tables.

Table 2-7: C/C++ Mode Selection Switches

Switch Name	*Description*
`-c89`	Supports programs that conform to the ISO/IEC 9899:1990 standard.
`-c99`	Supports programs that conform to the ISO/IEC 9899:1999 standard. It is the default mode.

Table 2-7: C/C++ Mode Selection Switches (Continued)

Switch Name	*Description*
`-c++`	Supports ISO/IEC 14882:2003 C++ standard with Analog Devices extensions.
`-c++11`	Supports ISO/IEC 14882:2011 C++ standard with Analog Devices extensions.
`-g++`	Enables GNU C++ features and extensions.

Table 2-8: C/C++ Compiler Common Switches

Switch Name	*Description*
`sourcefile`	Specifies file to be compiled.
`-@` *`filename`*	Reads command-line input from the file.
`-A` *`name[tokens]`*	Asserts the specified name as a predicate.
`-absolute-path-dependencies`	Uses absolute paths for the make dependencies emitted when using the `-M`, `-MM`, or `-MD` switches.
`-add-debug-libpaths`	Links against debug-specific variants of system libraries, where available.
`-alttok`	Allows alternative keywords and sequences in sources.
`-always-inline`	Treats `inline` keyword as a requirement rather than a suggestion.
`-annotate`	Annotates compiler-produced assembly files.
`-annotate-loop-instr`	Provides more annotation information for the prolog, kernel, and epilog of a loop.
`-asms-safe-in-simd-for-loops`	Informs the compiler that `asm` statements are not a barrier to SIMD inside loops with the `SIMD_for` pragma.
`-auto-attrs`	Based on the files the compiler processes, directs the compiler to emit automatic attributes. Enabled by default.
`-build-lib`	Directs the librarian to build a library file.
`-C`	Retains preprocessor comments in the output file; must run with the `-E` or `-P` switch.
`-c`	Compiles and/or assembles only, but does not link.
`-char-size[-8|-32]`	Selects 8-bit or 32-bit format for `char`, and 16-bit or 32-bit format for `short` respectively. The `-char-size-8` switch is only supported when compiling for processors that support byte-addressing, shown in the ***Processors Supporting Byte-Addressing*** table in the Processor Features. The `-char-size-32` switch is the default mode for processors that do not support byte-addressing.The `-char-size-8` switch is the default for processors that do support byte-addressing.
`-compatible-pm-dm`	Directs the compiler to treat `dm` and `pm` qualified pointers as assignment-compatible.
`-component` *`file.xml`*	Reads more options from the specified XML file.

Table 2-8: C/C++ Compiler Common Switches (Continued)

Switch Name	*Description*
-const-read-write	Specifies that data accessed via a pointer to const data can be modified elsewhere.
-const-strings	Directs the compiler to mark string literals as const qualified.
-D *macro*[=*definition*]	Defines a macro.
-dependency-add-target *target*	Adds target to any emitted dependency information.
-double-size[-32|-64]	Selects 32-bit or 64-bit IEEE format for double. The -double-size-32 switch is the default mode.
-double-size-any	Indicates that the resulting object can be linked with objects built with any double size.
-dry	Displays, but does not perform, main driver actions (verbose dry run).
-dryrun	Displays, but does not perform, top-level driver actions (terse dry run).
-E	Preprocesses, but does not compile, the source file.
-ED	Preprocesses and sends all output to a file.
-EE	Preprocesses and compiles the source file.
-eh	Enables exception handling.
-enum-is-int	Ensures that the enum type is int. By default, enum can have a type larger than int.
-extra-keywords	Recognizes Analog Devices extensions to ANSI/ISO standards for C and C++ (default mode).
-extra-precision	Directs the compiler not to: • generate double-word memory accesses • substitute an instruction for another that could result in the truncation of a 40-bit floating point value.
-file-attr *name[=value]*	Adds the specified attribute name/value pair to the file or files being compiled.
-flags-{asm|compiler|ipa|lib|link|mem|prelink} *switch[, switch2[, ...]]*	Passes command-line switches through the compiler to other build tools.
-float-to-int	Uses a support library function to convert a float to an integer type.
-force-circbuf	Treats array references of the form array[i%n] as circular buffer operations.
-fp-associative	Treats floating-point multiply and addition as an associative.
-full-version	Displays the version number of the driver and any processes invoked by the driver.
-fx-contract	Sets the default mode of FX_CONTRACT to ON.
-fx-rounding-mode-biased	Sets the default mode of FX_ROUNDING_MODE to BIASED.
-fx-rounding-mode-truncation	Sets the default mode of FX_ROUNDING_MODE to TRUNCATION.
-fx-rounding-mode-unbiased	Sets the default mode of FX_ROUNDING_MODE to UNBIASED.

Table 2-8: C/C++ Compiler Common Switches (Continued)

Switch Name	*Description*
`-g`	Generates DWARF-2 debug information.
`-glite`	Generates lightweight DWARF-2 debug information.
`-gnu-style-dependencies`	Produces dependency information in the style expected by the GNU `make` program.
`-H`	Outputs a list of included header files, but does not compile.
`-HH`	Outputs a list of included header files and compiles.
`-h[elp]`	Outputs a list of command-line switches.
`-I` *`directory[{,|;} directory...]`*	Appends directory to the standard search path.
`-I-`	Establishes the point in the `include` directory list at which the search for header files enclosed in angle brackets begins.
`-i`	Outputs only header details or makefile dependencies for `include` files specified in double quotes.
`-include` *`filename`*	Includes named file before preprocessing each source file.
`-ipa`	Enables interprocedural analysis.
`-L` *`directory[{;|,} directory …]`*	Appends directory to the standard library search path.
`-l` *`library`*	Searches library for functions when linking.
`-linear-simd`	Directs the compiler to attempt to generate SIMD code in linear code.
`-list-workarounds`	Lists all compiler-supported errata workarounds.
`-loop-simd`	Directs the compiler to attempt to generate SIMD code in loops.
`-M`	Generates make rules only, but does not compile.
`-MD`	Generates `make` rules, compiles, and prints to a file.
`-MM`	Generates make rules and compiles.
`-Mo` *`filename`*	Writes dependency information to *`filename`*. This switch is used with the `-ED` or `-MD` switch.
`-Mt` *`name`*	Makes dependencies, where the target is renamed as *`filename`*.
`-map` *`filename`*	Directs the linker to generate a memory map of all symbols.
`-mem`	Enables memory initialization.
`-multiline`	Enables string literals over multiple lines (default).
`-never-inline`	Ignores `inline` keyword on function definitions.
`-no-aligned-stack`	Does not attempt to maintain alignment of the program stack.
`-no-alttok`	Does not allow alternative keywords and sequences in sources.
`-no-annotate`	Disables the annotation of assembly files.

Table 2-8: C/C++ Compiler Common Switches (Continued)

Switch Name	*Description*
-no-annotate-loop-instr	Disables the production of extra loop annotation information by the compiler (default mode).
-no-assume-vols-are-iops	Instructs the compiler not to assume that volatile loads and stores are to IOP addresses.
-no-auto-attrs	Directs the compiler not to emit automatic attributes based on the files it compiles.
-no-circbuf	Disables the automatic generation of circular buffer code by the compiler.
-no-const-strings	Directs the compiler not to make string literals const qualified.
-no-db	Directs the compiler not to generate code containing delayed branches jumps.
-no-defs	Disables preprocessor definitions: macros, include directories, library directories, or keyword extensions.
-no-eh	Disables exception handling.
-no-extra-keywords	Disables Analog Devices keyword extensions that could be valid C/C++ identifiers.
-no-fp-associative	Does not treat floating-point multiply and addition as associative.
-no-fx-contract	Sets the default mode of FX_CONTRACT to OFF.
-no-linear-simd	Directs the compiler not to attempt to generate SIMD instructions in linear code.
-no-loop-simd	Directs the compiler not to attempt to generate SIMD instructions in loop code without the SIMD_for or vector_for pragma.
-no-main-calls-exit	Prevents the compiler from inserting a call to exit() at the end of main().
-no-mem	Disables memory initialization.
-no-multiline	Disables multiple line string literal support.
-no-progress-rep-timeout	Prevents the compiler from issuing a diagnostic during excessively long compilations.
-no-rtcheck	Disables run-time checking.
-no-rtcheck-arr-bnd	Disables checking array boundaries at run time.
-no-rtcheck-div-zero	Disables checking for division by zero at run time.
-no-rtcheck-heap	Disables checking of heap operations zero at run time.
-no-rtcheck-null-ptr	Disables checking for NULL pointer dereferences at run time.
-no-rtcheck-shift-check	Disables checking for negative/too-large shifts at run-time.
-no-rtcheck-stack	Disables checking for stack overflow at run time.
-no-rtcheck-unassigned	Disables checking for unassigned variables at run time.
-no-sat-associative	Saturating addition is not associative.

Table 2-8: C/C++ Compiler Common Switches (Continued)

Switch Name	*Description*
`-no-saturation`	Causes the compiler not to introduce saturation semantics when optimizing expressions.
`-no-shift-to-add`	Directs the compiler not to replace an arithmetic shift by one with an add instruction.
`-no-simd`	Disables automatic SIMD mode.
`-no-std-ass`	Disables any predefined assertions and system-specific macro definitions.
`-no-std-def`	Disables preprocessor definitions and Analog Devices keyword extensions that do not have leading underscores (__).
`-no-std-inc`	Searches for preprocessor include header files only in the current directory and in directories specified with the `-I` switch.
`-no-std-lib`	Searches for only those library files specified with the `-l` switch.
`-no-threads`	Specifies that all compiled code can be non-thread-safe.
`-no-workaround` *workaround_id[, workaround_id …]*	Disables specific hardware anomaly workarounds within the compiler.
`-normal-word-code`	Directs the compiler to generate instructions of normal-word size (48-bits).
`-nwc`	Has the same effect as compiling with the `-normal-word-code` switch.
`-O[0|1]`	Enables code optimizations.
`-Oa`	Enables automatic function inlining.
`-Os`	Optimizes for code size.
`-Ov` *num*	Controls speed versus size optimizations.
`-o` *filename*	Specifies the output file name.
`-overlay`	Disables the propagation of register information between functions and forces the compiler to assume that all functions clobber all scratch registers.
`-overlay-clobbers` *clobbered-regs*	Specifies the registers for an overlay manager to clobber.
`-P`	Preprocesses, but does not compile, the source file. Omits line numbers in the preprocessor output.
`-PP`	Similar to `-P`, but does not halt compilation after preprocessing.
`-p`	Generates profiling instrumentation.
`-path-{ asm | compiler | ipa | lib | link | prelink }` *pathname*	Uses the specified pathname as the location of the specified compilation tool (assembler, compiler, IPA solver, library builder, linker, or prelinker, respectively).
`-path-install` *directory*	Uses the specified directory as the location of all compilation tools.
`-path-output` *directory*	Specifies the location of non-temporary files.
`-path-temp` *directory*	Specifies the location of temporary files.
`-pgo-session` *session-id*	Used with profile-guided optimization.

Table 2-8: C/C++ Compiler Common Switches (Continued)

Switch Name	*Description*
`-pguide`	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization.
`-pplist` *filename*	Outputs a raw preprocessed listing to the specified file.
`-proc` *processor*	Directs the compiler to produce code suitable for the specified processor.
`-prof-hw`	Directs the compiler to generate profiling code targeted for execution on hardware. Requires the use of a supported profiling switch.
`-progress-rep-func`	Issues a diagnostic message each time the compiler starts compiling a new function. Equivalent to `-Wwarn=cc1472`.
`-progress-rep-opt`	Issues a diagnostic message each time the compiler starts a new generic optimization pass on the current function. Equivalent to `-Wwarn=cc1473`.
`-progress-rep-timeout`	Issues a diagnostic message if the compiler exceeds a time limit during compilation.
`-progress-rep-timeout-secs` *secs*	Specifies how many seconds must elapse during a compilation before the compiler issues a diagnostic on the length of compilation.
`-R` *directory[{:|,} directory …]*	Appends directory to the standard search path for source files.
`-R-`	Removes all directories from the standard search path for source files.
`-reserve` *register[, register …]*	Reserves certain registers from compiler use. *Note:* Reserving registers can have a detrimental effect on optimization capabilities of the compiler.
`-restrict-hardware-loops` *maximum*	Restricts the number of levels of loop nesting used by the compiler.
`-rtcheck`	Enables run-time checking.
`-rtcheck-arr-bnd`	Enables run-time checking of array boundaries.
`-rtcheck-div-zero`	Enables run-time checking for division by zero.
`-rtcheck-heap`	Enables run-time checking of heap operations.
`-rtcheck-null-ptr`	Enables run-time checking for NULL pointer dereferences.
`-rtcheck-shift-check`	Enables run-time checking for negative/too-large shifts.
`-rtcheck-stack`	Enables run-time checking for stack overflow.
`-rtcheck-unassigned`	Enables run-time checking for unassigned variables.
`-S`	Stops compilation before running the assembler.
`-s`	Removes debug info from the output executable file.
`-sat-associative`	Saturating addition is associative.
`-save-temps`	Saves intermediate files.
`-section`*id=section_name[, id=section_name...]*	Orders the compiler to place data/program of type *`id`* into the *`section_name`* section.

Table 2-8: C/C++ Compiler Common Switches (Continued)

Switch Name	*Description*
`-short-word-code`	Directs the compiler to generate instructions of short word size (16/32/48-bits).
`-show`	Displays the driver command-line information.
`-si-revision` *`version`*	Specifies a silicon revision of the specified processor. The default setting is the latest silicon revision at the time of release.
`-signed-bitfield`	Makes the default type for `int` bit-fields signed.
`-structs-do-not-overlap`	Specifies that `struct` copies can use `memcpy` semantics, rather than the usual `memmove` behavior.
`-swc`	Directs the compiler to generate instructions of short word size (16/32/48-bits).
`-syntax-only`	Checks the source code for compiler syntax errors, but does not write any output.
`-sysdefs`	Defines the system definition macros.
`-T` *`filename`*	Specifies the Linker Description File.
`-threads`	Specifies that support for multithreaded applications is enabled.
`-time`	Displays the elapsed time as part of the output information on each part of the compilation process.
`-U` *`macro`*	Undefines macro.
`-unsigned-bitfield`	Makes the default type for plain `int` bit-fields unsigned.
`-v`	Displays both the version and command-line information.
`-verbose`	Displays command-line information.
`-version`	Displays version information.
`-W{annotation|error|remark|suppress|warn}` *`number[, number ...]`*	Overrides the default severity of the specified error message.
`-Wannotations`	Indicates that the compiler can issue code-generation annotations. Annotations are messages milder than warnings and can help to optimize your code.
`-Werror-limit` *`number`*	Stops compiling after reaching the specified number of errors.
`-Werror-warnings`	Directs the compiler to treat all warnings as errors.
`-Wremarks`	Indicates that the compiler can issue remarks. Remarks are diagnostic messages milder than warnings.
`-Wterse`	Issues only the briefest form of compiler warning, errors, and remarks.
`-w`	Does not display compiler warning messages.
`-warn-component`	Issues warnings if any libraries specified by component XML files could not be located.
`-warn-protos`	Produces a warning when a function is called without a prototype.
`-workaround` *`workaround_id[, workaround_id…]`*	Enables code-generator workaround for specific hardware errata.

Table 2-8: C/C++ Compiler Common Switches (Continued)

Switch Name	*Description*
`-xref` *`filename`*	Outputs cross-reference information to the specified file.

Table 2-9: C Mode (MISRA) Compiler Switches

Switch Name	Description
`-misra`	Enables checking for MISRA-C: 2004 guidelines, allows some relaxation of interpretation.
`-misra-linkdir` *`directory`*	Specifies directory for generation of `.misra` files. If this option is not specified, a local directory called `MISRARepository` is created.
`-misra-no-cross-module`	Enables checking for MISRA-C: 2004 guidelines, allows some relaxation of interpretation. Does not generate `.misra` files to check for link-time rule violations.
`-misra-no-runtime`	Enables checking for MISRA-C: 2004 guidelines, allows some relaxation of interpretation. Does not generate extra code to perform run-time checking in support of a number of Rules.
`-misra-strict`	Enables checking for MISRA-C: 2004 guidelines
`-misra-suppress-advisory`	Enables checking for MISRA-C: 2004 guidelines. Advisory rules are not reported.
`-misra-testing`	Enables checking for MISRA-C: 2004 guidelines. Suppresses reporting of MISRA-C rule 20.4, 20.7, 20.8, 20.9, 20.10, 20.11, and 20.12.
`-Wmis_suppress` *`rule_number[, rule_number]`*	Overrides the default severity of the specified messages relating to the specified MISRA-C rules.
`-Wmis_warn` *`rule_number[, rule_number]`*	Overrides the default severity of the specified messages relating to the specified MISRA-C rules.

Table 2-10: C++ Mode Compiler Switches

Switch Name	Description
`-anach`	Supports some language features (anachronisms) prohibited by the C++ standard but still in common use.
`-check-init-order`	Adds run-time checking to the generated code, highlighting potential uninitialized external objects.
`-friend-injection`	Allows non-standard behavior concerning friend declarations. When friend names are not injected, function names are visible only when using argument-dependent lookup.
`-full-dependency-inclusion`	Ensures reinclusion of implicitly included files when generating dependency information.
`-implicit-inclusion`	Allows implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. It is not compatible with exported templates.
`-no-anach`	Disallows the use of anachronisms prohibited by the C++ standard.
`-no-friend-injection`	Allows standard behavior. Friend function names are visible only when using argument-dependent lookup. Friend class names are never visible.

Table 2-10: C++ Mode Compiler Switches (Continued)

Switch Name	Description
`-no-implicit-inclusion`	Prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.
`-no-rtti`	Disables run-time type information.
`-no-std-templates`	Disables the lookup of names used in templates.
`-rtti`	Enables run-time type information.
`-std-templates`	Enables the lookup of names used in templates.

C/C++ Mode Selection Switch Descriptions

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

-c89

The `-c89` switch directs the compiler to support programs that conform to the ISO/IEC 9899:1990 standard. For greater conformance to the standard, use the following switches: `-alttok`, `-const-read-write`, and `-no-extra-keywords`. See the ***C/C++ Compiler Common Switches*** table in Compiler Command-Line Switches.

-c99

The `-c99` switch directs the compiler to support programs that conform to a freestanding implementation of the ISO/IEC 9899:1999 standard. For greater conformance to the standard, use the following switches: `-alttok`, `-const-read-write`, and `-no-extra-keywords`. See the ***C/C++ Compiler Common Switches*** table in Compiler Command-Line Switches.

NOTE: The compiler supports the `_Complex` keyword but not the `_Imaginary` keyword.

-c++

The `-c++` (C++ mode) switch directs the compiler to compile the source file(s) written in ISO/IEC 14882:2003 standard C++ with Analog Devices language extensions. When using this switch, source files with an extension of `.c` are compiled and linked in C++ mode. The compiler implicitly adds this switch when compiling files with a `.cpp` extension.

All C++ standard features are accepted in the default mode, except exception handling and run-time type identification. The exception handling and run-time type identification 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 the ***C++ Mode Compiler Switches*** table in Compiler Command-Line Switches.

-c++11

The `-c++11` (C++11 mode) switch directs the compiler to compile the source files written in ISO/IEC 14882:2011 standard C++ with Analog Devices language extensions. When using this switch, source files with an extension of `.c` are compiled and linked in C++ mode.

This version of the compiler accepts many of the ISO/IEC 14882:2011 standard features, but the underlying library support conforms to the ISO/IEC 14882:2003 standard. Enable exception handling and run-time type

identification explicitly, because these features 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 the ***C++ Mode Compiler Switches*** table in Compiler Command-Line Switches.

`-g++`

The `-g++` (G++ mode) switch directs the compiler to emulate many extensions available in GNU C++. By default, the C++ dialect is based on the ISO/IEC 14882:2003 standard. To base the dialect on the ISOIEC 1482:2011 standard instead, enable the additional `-c++11` switch.

You must enable exception handling and run-time type identification explicitly if you require these features, because they impose a run-time overhead that is not desirable for all embedded programs. Use the `-eh` and `-rtti` switches to do so. See the ***C++ Mode Compiler Switches*** table in Compiler Command-Line Switches.

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. The ***Input and Output File Extensions*** and ***File Extensions Specifying Compiler Action*** tables (in Running the Compiler) list the permitted extensions and matching compiler operations.

`-@` ***`filename`***

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

`-A` ***`name[tokens]`***

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

Table 2-11: Predefined Assertions

Assertion	*Value*
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 can 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 can be tested as:

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

NOTE: 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.

-absolute-path-dependencies

The `-absolute-path-dependencies` switch can be used with one of the `-M`, `-MM`, or `-MD` switches. By default, these switches emit make dependencies using relative paths. The `-absolute-path-dependencies` switch can be used to emit the dependencies using absolute, rather than relative paths. The compiler driver invokes all underlying tools with the switch as required, so that all tools emit dependencies using absolute paths. The underlying tools are the compiler, assembler, preprocessor, linker, and archiver.

-add-debug-libpaths

The `-add-debug-libpaths` switch prepends the `Debug` subdirectory to the search paths passed to the linker. The `Debug` subdirectory, found in each of the silicon-revision-specific library directories, contains variants of certain libraries. For example, system services, which provide extra diagnostic output to help with debugging problems arising from their use.

NOTE: Invoke this switch from the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Linker*** > ***Processor*** > ***Use Debug System libraries***.

-alttok

In C89 and C99 modes, the `-alttok` (alternative tokens) switch directs the compiler to allow digraph sequences in source files. This switch is enabled by default in C89 and C99 modes.

In C++ mode, this switch is disabled by default. When enabled in C++ mode, the switch also enables the recognition of alternative operator keywords in C++ source files. The ***Alternative Operator Keywords*** table lists the keywords.

Table 2-12: Alternative Operator Keywords

Keyword	*Equivalent*
`and`	`&&`
`and_eq`	`&=`
`bitand`	`&`
`bitor`	`|`

Table 2-12: Alternative Operator Keywords (Continued)

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

See also `-no-alttok`.

NOTE: The `-alttok` switch has no effect on the use of the alternative tokens listed in the ***Alternative Operator Keywords*** table in C89 or C99 mode. Instead, in C89 or C99 mode, include header file `<iso646.h>` to use alternative tokens.

-always-inline

The `-always-inline` switch instructs the compiler always to attempt inlining any call to a function that is defined with the `inline` qualifier. It is equivalent to applying #pragma always_inline to all functions in the module that have the `inline` qualifier.

See also `-never-inline`.

-annotate

The `-annotate` (enable assembly annotations) switch directs the compiler to annotate assembly files generated by the compiler. The default behavior is that whenever optimizations are enabled, all assembly files generated by the compiler are annotated. The files are annotated with information on the performance of the generated assembly.

For more information, see Assembly Optimizer Annotations in the Optimal Performance from C/C++ Source Code chapter.

See also `-no-annotate`.

NOTE: Invoke this switch from the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Generate annotations***.

-annotate-loop-instr

The `-annotate-loop-instr` switch directs the compiler to provide more annotation information for the prolog, kernel, and epilog of a loop. See Assembly Optimizer Annotations in the Optimal Performance from C/C++ Source Code chapter. See also `-no-annotate-loop-instr`.

-asms-safe-in-simd-for-loops

The `-asms-safe-in-simd-for-loops` switch directs the compiler not to consider `asm` statements within loops a barrier to SIMD code generation if the loop is decorated with a `SIMD_for` pragma.

See also SIMD Support.

-auto-attrs

The -auto-attrs (automatic attributes) switch directs the compiler to emit automatic attributes based on the files it compiles. Emission of automatic attributes is enabled by default. See File Attributes for more information about attributes, including automatic attributes that the compiler emits. See also the -no-auto-attrs and -file-attr *name[=value]* switches.

> **NOTE:** Invoke this switch from the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Auto-generated attributes***.

-build-lib

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

-C

The -C (comments) switch directs the C/C++ preprocessor to retain comments in its output file. The -C switch must be used with the -E or -P switch.

-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 (.doj) file for each source file.

-char-size[-8|-32]

The -char-size-8 (char is 8 bits, short is 16 bits) and -char-size-32 (both char and short are 32 bits) switches select the storage format that the compiler uses for types char and short. The switch also affects the endianness of 64-bit integer and floating-point types, as well as a number of other changes to the run-time model. The -char-size-8 switch only is supported for processors that support byte-addressing, and for those processors, it is the default setting. The ***Processors Supporting Byte-Addressing*** table in Processor Features identifies these processors. For processors without byte-addressing support, the default (and only) mode is -char-size-32. For more information, see Using Byte-Addressing.

The -char-size-8 switch defines the __BYTE_ADDRESSING__ macro, while the -char-size-32 switch undefines the __BYTE_ADDRESSING__ macro.

> **NOTE:** Invoke this switch in the IDE by setting ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor*** > ***Char size*** to the required value.

-compatible-pm-dm

The compatible-pm-dm switch directs the compiler to treat dm and pm qualified pointers as assignment-compatible.

-component *file.xml*

The `-component` (read component file) switch instructs the compiler to read the specified *`file.xml`*. The switch also instructs the compiler to retrieve additional switches while building applications using the component. The IDE uses this switch to build projects that employ add-in products in addition to CCES.

See also `-warn-component`.

-const-read-write

The `-const-read-write` switch specifies to the compiler that constants can 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 behavior to match the ANSI C assumption, namely that the data may be changed via other non-`const` pointers.

> **NOTE:** Invoke this switch in the IDE via *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Language Settings* > *Pointers to const may point to non-const data.*

-const-strings

The `-const-strings` (`const`-qualify strings) switch directs the compiler to mark string literals as `const`-qualified. This switch is the default behavior.

See also `-no-const-strings` .

> **NOTE:** Invoke this switch in the IDE via *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Language Settings* > *Literal strings are const.*

-D *macro*[=*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'`. The compiler processes all `-D` switches on the command line before any `-U` (undefine macro) switches.

See also `-U` *`macro`*.

> **NOTE:** Add instances of this switch in the IDE via *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Preprocessor* > *Preprocessor definitions.*

-dependency-add-target *target*

The `-dependency-add-target` switch adds *target* as another target that relies on the dependencies in the build. Use this switch with switches for emitting dependency information, for example, `-M`.

For example, if you are building `apple.doj` from `apple.c`, the compiler dependency output indicates that `apple.doj` depends on `apple.c`. Using `-dependency-add-target pear.doj` causes the compiler to emit extra dependency information to indicate that `pear.doj` also depends on `apple.c`.

-double-size[-32|-64]

The `-double-size-32` (`double` is 32 bits) and `-double-size-64` (double is 64 bits) switches select the storage format that the compiler uses for type `double`. The default mode is `-double-size-32`. For more information, see Using Data Storage Formats .

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

> **NOTE:** Invoke this switch in the IDE by setting ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor*** > ***Double size*** to the required value.

-double-size-any

The `-double-size-any` switch specifies that the input source files make no use of `double` type data. It also specifies that the resulting object files to be marked in a way that enables them to be linked against objects built with `doubles`, either 32- or 64-bit in size.

> **NOTE:** Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor*** > ***Allow mixing of sizes***.

-dry

The `-dry` (verbose dry run) switch directs the compiler to display main `cc21k` actions, but not to perform them.

-dryrun

The `-dryrun` (terse dry run) switch directs the compiler to display top-level `cc21k` actions, but not to perform them.

-E

The `-E` (stop after preprocessing) switch directs the compiler to stop after the C/C++ preprocessor runs (without compiling). The output (preprocessed source code) prints to the standard output stream unless the output file is specified with the `-o` switch.

-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.

> **NOTE:** Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Generate preprocessed file***.

-EE

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

-eh

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

If used when compiling C programs, without the -c++ (C++ mode) switch, the -eh switch directs the compiler to generate exceptions tables but does not change the language accepted. In this case, the __EXCEPTIONS macro is not defined.

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

Object files created with exceptions enabled can 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. If an attempt is made to throw an exception through the execution of a function not compiled with -eh, then abort or the function registered with set_terminate is called. See also #pragma generate_exceptions_tables and -no-eh.

In non-threaded applications, the buffer used for the passing of exception data is not returned to the heap on application exit. The buffer is not returned to avoid unnecessary code and has no impact on behavior.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Language Settings*** > ***C++ exceptions and RTTI.***

-enum-is-int

The -enum-is-int switch ensures that the type of an enum is int. By default, the compiler defines enumeration types with integral types larger than int, if int is insufficient to represent all the values in the enumeration. This switch prevents the compiler from selecting a type wider than int. See Enumeration Type Implementation Details for more information.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Language Settings*** > ***Enumerated types are always int.***

2.2.3.2.31 -extra-keywords

-extra-precision

The -extra-precision switch directs the compiler to avoid instructions and code sequences that can work incorrectly if 40-bit memory accesses are enabled. When the switch is used, the compiler does not generate the following:

- Long word (LW) memory accesses
- SIMD code (unless one of the loop optimization pragmas, SIMD_for or vector_for, is used)
- Instructions that can result in 40-bit values being truncated. For example, to aid instruction grouping, the compiler can replace a register transfer instruction with an integer PASS instruction. Although this gives more opportunities for issuing parallel instructions, it can result in the truncation of a 40-bit extended precision floating point value to a 32-bit floating point value.

-file-attr *name[=value]*

The `-file-attr` (file attribute) switch directs the compiler to add the specified attribute name/value pair to all of the files it compiles. To add multiple attributes, use the switch multiple times. If `=value` is omitted, the default value of "1" is used. See File Attributes for more information about attributes, including automatic attributes the compiler emits.

See also `-auto-attrs` and `-no-auto-attrs`.

> **NOTE:** Add instances of this switch in the IDE via *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *General* > *Additional attributes*.

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

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

The tools are listed in the ***Switches Passed to Other Build Tools*** table.

Table 2-13: Switches Passed to Other Build Tools

Switch	*Tool*
`-asm`	Assembler
`-compiler`	Compiler executable
`-ipa`	IPA solver
`-lib`	Library builder (`elfar.exe`)
`-link`	Linker
`-mem`	Memory initializer
`-prelink`	Prelinker

-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 use circular buffer facilities. The circular buffer facilities are used 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 default behavior, regarding circular buffers, is conservative. In default mode, circular buffers are not used unless verified that the circular index or pointer is always within the circular buffer range. See Circular Buffer Built-In Functions.

> **NOTE:** Invoke this switch in the IDE by setting *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Language Settings* > *Circular buffer generation* to *Even when pointer may be outside buffer range (-force-circbuf)*.

-fp-associative

The -fp-associativeswitch permits the compiler to treat floating-point addition and multiplication as associative operations, allowing it to reorder sequences of additions and multiplications where that is beneficial for performance. Due to different rounding of intermediate results, this may change the result of such sequences.

This switch is on by default if optimization is enabled.

See also -no-fp-associative .

-full-version

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

-fx-contract

The -fx-contract switch sets the default state of FX_CONTRACT to ON, which is the default setting. This switch controls the performance and accuracy of arithmetic on the native fixed-point type, fract. See FX_CONTRACT for more information.

See also -no-fx-contract.

-fx-rounding-mode-biased

The -fx-rounding-mode-biased switch sets the default state of FX_ROUNDING_MODE to BIASED. This switch controls the rounding behavior of arithmetic on the native fixed-point type, fract. See Setting the Rounding Mode for more information.

-fx-rounding-mode-truncation

The -fx-rounding-mode-truncation switch sets the default state of FX_ROUNDING_MODE to TRUNCATION, which is the default setting. This switch controls the rounding behavior of arithmetic on the native fixed-point type, fract. See Setting the Rounding Mode for more information.

-fx-rounding-mode-unbiased

The -fx-rounding-mode-unbiased switch sets the default state of FX_ROUNDING_MODE to UNBIASED. This switch controls the rounding behavior of arithmetic on the native fixed-point type, fract. See Setting the Rounding Mode for more information.

-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 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 CCES IDE (debugger). This combination of switches provides line debugging and global variable debugging.

NOTE: When the -g and -O switches are specified, no debug information is available for local variables. Also, the standard optimizations can rearrange program code such that inaccurate line number information is produced. For full debugging capabilities, use the -g switch without the -O switch.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Generate debug information***.

-glite

The `-glite` (lightweight debugging) switch can be used on its own, or with the `-g` compiler switch. The switch instructs the compiler to remove any unnecessary debug information from the compiled code.

When used on its own, the switch also enables the `-g` switch.

NOTE: Use this switch to reduce the size of object and executable files. The switch has no effect on the size of the code loaded onto the target.

-gnu-style-dependencies

The `-gnu-style-dependencies` switch changes the format in which dependency information, such as that produced by the `-M` switch, is produced, so that it matches the format used by the GNU `make` program. The ***Effect of -gnu-style-dependencies Switch*** table shows the switch effect.

Table 2-14: Effect of -gnu-style-dependencies Switch

	Without **-gnu-style-dependencies**	*With* **-gnu-style-dependencies**
Quoting	Yes (`"foo"`)	No (`foo`)
Whitespace	Quoted (`"x y"`)	Escaped with backslash (`x\ y`)
Directory separators	Backslash (`\`)	Forward slash (`/`)
Path form	Canonical (`"c:\foo\bar"`)	Relative (`../bar`)

The IDE applies this switch automatically.

-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 a list of the files included by the preprocessor via the `#include` directive. The list is output to the standard output stream. 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.

NOTE: Add instances of this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Preprocessor*** > ***Additional include directories***.

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, `<CCES_install_dir>\...\include`

NOTE: 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 at which the search for header files enclosed in angle brackets begins. The point is in the `include` directory list Normally, for header files enclosed in double quotes, the compiler searches in the directory containing the current input file. Then the compiler reverts to looking in the directories specified with the `-I-` switch, then in the standard include directory.

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

NOTE: The `-I-` switch removes the directory containing the current input file from the include directory list.

-i

The `-i` (less includes) switch can be used with the `-H`, `-HH`, `-M`, and `-MM` switches. The switch directs the compiler to output only header details (`-H`, `-HH`) or makefile dependencies (`-M`, `-MM`) for `include` files 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` switches on the command line are processed before an `-include` file. Only one `-include` can 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. If used, the `-ipa` option should be applied to all C and C++ files in the program. For more information, see Interprocedural Analysis. Specifying `-ipa` also implies setting the `-O[0|1]` switch.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Interprocedural optimization***.

-L *directory[{;|,} directory …]*

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

NOTE: Add instances of this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Linker*** > ***General*** > ***Search directories***.

-l *library*

The -l (link library) switch directs the compiler 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.

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

If one of the definitions for x comes from user objects, and the other 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.

NOTE: Add instances of this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Linker*** > ***General*** > ***Additional libraries and object files***.

-linear-simd

The -linear-simd (generate linear SIMD code) switch directs the compiler to attempt to produce SIMD instructions in linear code. For more information, see SIMD Support.

See also -no-linear-simd.

-list-workarounds

The -list-workarounds (list supported errata workarounds) switch displays a list of all errata workarounds which the compiler supports. See Controlling Silicon Revision and Anomaly Workarounds Within the Compiler for details of valid workarounds and the interaction of the -si-revision, -workaround, and -no-workaround switches.

-loop-simd

The -loop-simd (use SIMD in loops) switch directs the compiler to generate SIMD code when possible.

ATTENTION: On certain SHARC processors (e.g. ADSP-21367/8/9), SIMD accesses of external memory are not supported. Do not use this switch if the data that is accessed in the loop might be mapped to external memory. For more information, refer to your processor's hardware reference manual.

See also `-no-loop-simd` , SIMD Support.

-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 `-Mo` *`filename`*.

-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 *name*

The `-Mt` *`name`* (output make rule for the named source) switch modifies the target of generated dependencies, renaming the target to name. It only has an effect when used in conjunction with the `-M` or `-MM` switch.

-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 (utility). The memory initializer can be controlled through the `mem` switch, or disabled using the `-no-mem` switch.

For more information, see:

- ***Processor Startup*** in the *System Run-Time Documentation*.
- ***Memory Initializer*** in the *Linker and Utilities Manual*.

-multiline

The `-multiline` switch directs the compiler to allow string literals to span multiple lines without the need for a "\" at the end of each line. This is the default mode for C source files. The switch has no effect when compiling C++ source files which do not support multiline string literals.

See also `-no-multiline`.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Language Settings*** > ***Allow multi-line character strings***.

-never-inline

The `-never-inline` switch instructs the compiler to ignore the `inline` qualifier on function definitions, so that no calls to such functions will be inlined.

See also `-always-inline`.

NOTE: Invoke this switch in the IDE by setting ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Inlining to Never***.

-no-aligned-stack

The `-no-aligned-stack` (disable stack alignment) switch directs the compiler to not to attempt to maintain alignment of the program stack on a double-word boundary. This can result in slightly more efficient code and use of stack space if there is no requirement to keep the stack aligned. See Stack Alignment for more information.

-no-alttok

The `-no-alttok` (disable alternative tokens) switch directs the compiler not to accept digraph sequences in the source files. This switch is enabled by default in C++ mode, and disabled by default in C89 and C99 modes. In C++ mode, the switch also controls the acceptance of alternative operator keywords.

See also `-alttok`.

-no-annotate

The `-no-annotate` (disable assembly annotations) switch directs the compiler not to annotate assembly files generated by the compiler. 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 in the Optimal Performance from C/C++ Source Code chapter for more details on this feature.

See also `-annotate`.

NOTE: Invoke this switch in the IDE by clearing ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Generate annotations***.

-no-annotate-loop-instr

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

See also `-annotate-loop-instr`.

-no-assume-vols-are-iops

The `-no-assume-vols-are-iops` switch specifies that the compiler should not assume that volatile accesses to memory are accessing memory-mapped IOP registers. By default, the compiler will apply workarounds for IOP-related anomalies to volatile memory accesses, unless it can determine that the access is not to an IOP register. When the `-no-assume-vols-are-iops` switch is used, memory-mapped IOP registers should be accessed using the `iop_read` and `iop_write` built-in functions (see Miscellaneous Built-In Functions).

-no-auto-attrs

The `-no-auto-attrs` (no automatic attributes) switch directs the compiler not to emit automatic attributes based on the files it compiles. Emission of automatic attributes is enabled by default. See File Attributes for more information about attributes, and what automatic attributes the compiler emits.

See also `-auto-attrs` and `-file-attr` *`name[=value]`*.

> **NOTE:** Invoke this switch in the IDE by clearing ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Auto-generated attributes***.

-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.

> **NOTE:** Invoke this switch in the IDE by setting ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Language Settings*** > ***Circular Buffer Generation*** to ***Never***.

-no-const-strings

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

See also `-const-strings`.

> **NOTE:** Invoke this switch in the IDE by clearing ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Language Settings*** > ***Literal strings are const***.

-no-db

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

> **NOTE:** Disabling of interrupts within the epilogue code of a re-entrant interrupt function still uses a delayed branch jump to minimize interrupt latency.

-no-defs

The `-no-defs` (disable defaults) switch directs the preprocessor not to define any default preprocessor macros, include directories, library directories or libraries. It also disables the Analog Devices `cc21k` C/C++ keyword extensions.

-no-eh

The `-no-eh` (disable exception handling) switch directs the compiler to disallow ANSI/ISO C++ exception handling. This is the default mode.

See also `-eh`.

-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 conflict with valid C/C++ identifiers, for example, keywords such as `pm` and `dm`. Alternate keywords, which are prefixed with two leading underscores, such as `__pm` and `__dm`, continue to work.

See also `-extra-keywords`.

> **NOTE:** Invoke this switch in the IDE via *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Language Settings* > *Disable Analog Devices extension keywords*.

-no-fp-associative

By default, when optimization is enabled, the compiler may treat floating-point addition and multiplication as associative operations, to allow sequences of additions or multiplications to be reordered where that is beneficial for performance. Due to different rounding of intermediate results, this may change the result of such sequences.

The `-no-fp-associative`switch diables this behavior.

See also `-fp-associative` .

> **NOTE:** Invoke this switch in the IDE via *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Language Settings* > *Do not treat floating-point operations as associative*.

-no-fx-contract

The `-no-fx-contract` switch sets the default state of `FX_CONTRACT` to `OFF`. This switch controls the performance and accuracy of arithmetic on the native fixed-point type, `fract`. See FX_CONTRACT for more information.

See also `-fx-contract`.

-no-linear-simd

The `-no-linear-simd` (do not generate linear SIMD code) switch directs the compiler not to attempt to produce SIMD instructions in linear code. For more information, see SIMD Support.

See also `-linear-simd`.

-no-loop-simd

The `-no-loop-simd` (do not generate loop SIMD code) switch directs the compiler not to attempt to produce SIMD instructions in loops except those marked with the `SIMD_for` or `vector_for` pragmas. For more information, see SIMD Support.

See also `-loop-simd` .

-no-main-calls-exit

The -no-main-calls-exit switch specifies that the compiler should not insert a call to exit() at the end of main(). Instead, main() will end with the standard function return sequence.

-no-mem

The -no-mem (disable memory initialization) switch directs the compiler not to run the mem21k initializer.

See also -mem.

-no-multiline

The -no-multiline switch directs the compiler to disallow string literals which span multiple lines without a "\" at the end of each line.

See also -multiline .

NOTE: Invoke this switch by clearing ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Language Settings*** > ***Allow multi-line character strings***.

-no-progress-rep-timeout

The -no-progress-rep-timeout (disable progress message for long compilations) switch disables the diagnostic message issued by the compiler to indicate that it is still working, when a function's compilation is taking an excessively long time. The message is disabled by default.

See also -progress-rep-timeout and -progress-rep-timeout-secs *secs*.

-no-rtcheck

The -no-rtcheck (disable run-time checking) switch directs the compiler to disable generation of additional code to check at runtime for common programming errors. This switch is the default, and is equivalent to specifying all of the following switches:

- -no-rtcheck-arr-bnd
- -no-rtcheck-div-zero
- -no-rtcheck-heap
- -no-rtcheck-null-ptr
- -no-rtcheck-shift-check
- -no-rtcheck-stack
- -no-rtcheck-unassigned

See also -rtcheck.

-no-rtcheck-arr-bnd

The -no-rtcheck-arr-bnd (do not check array bounds at runtime) switch directs the compiler not to generate additional code to verify that array accesses are within the bounds of the array.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor*.

See also `-rtcheck-arr-bnd` and `-rtcheck`.

`-no-rtcheck-div-zero`

The `-no-rtcheck-div-zero` (do not check for division by zero at runtime) switch directs the compiler not to generate additional code to verify that divisors are non-zero before performing division operations.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor*.

See also `-rtcheck-div-zero` and `-rtcheck`.

`-no-rtcheck-heap`

The `-no-rtcheck-heap` (do not heap operations at runtime) switch directs the compiler not to link against the debugging version of the heap libraries.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor*.

See also `-rtcheck-heap` and `-rtcheck`.

`-no-rtcheck-null-ptr`

The `-no-rtcheck-null-ptr` (do not check for NULL pointers at runtime) switch directs the compiler not to generate additional code to verify that pointers are not NULL, before dereferencing them.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor*.

See also `-rtcheck-null-ptr` and `-rtcheck`.

`-no-rtcheck-shift-check`

The `-no-rtcheck-shift-check` (do not check shift values at runtime) switch directs the compiler not to generate additional code to verify that, when shifting a value X by some amount Y, Y is a positive amount, and less than the number of bits used to represent X's type.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor*.

See also `-rtcheck-shift-check` and `-rtcheck`.

`-no-rtcheck-stack`

The `-no-rtcheck-stack` (do not check for stack overflow at runtime) switch directs the compiler not to link in the modified startup and interrupt vector code to enable CB7I to trap stack overflow occurrences.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor*.

See also -rtcheck-stack and -rtcheck.

-no-rtcheck-unassigned

The -no-rtcheck-unassigned (do not check variables are assigned at runtime) switch directs the compiler not to generate additional code to verify that variables have been assigned a value before they are used.

NOTE: Invoke this behavior in the IDE via the run-time checking options under *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor*.

See also -rtcheck-unassigned and -rtcheck.

-no-sat-associative

The -no-sat-associative (saturating addition is not associative) switch instructs the compiler not to consider saturating addition operations as associative: (a+b)+c may not be rewritten as a+(b+c), when the addition operator saturates. The default is that saturating addition is not associative.

See also -sat-associative.

-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. The code produced may be less efficient than when the switch is not used. Saturation is enabled by default when optimizing, and may be disabled by this switch. Saturation is disabled when not optimizing (this switch is the default when not optimizing).

-no-shift-to-add

The -no-shift-to-add switch prevents the compiler from replacing a shift-by-one instruction with an addition. While this can produce faster code, it can also lead to arithmetic overflow.

NOTE: Invoke this switch in the IDE via *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor* > *Disable shift-to-add conversion*.

-no-simd

The -no-simd (disable SIMD mode) switch directs the compiler to disable automatic SIMD code generation. 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 also SIMD Support.

NOTE: Invoke this switch in the IDE via *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor* > *Disable automatic SIMD code generation*.

-no-std-ass

The -no-std-ass (disable standard assertions) switch prevents the compiler from defining the standard assertions.

See -A *name[tokens]* 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.

> **NOTE:** 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.

> **NOTE:** Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Preprocessor*** > ***Ignore standard include paths***.

-no-std-lib

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

-no-threads

The `-no-threads` (disable thread-safe build) switch directs the compiler to link against the non-thread-safe variants of the C/C++ run-time library. This is the default.

See also `-threads`.

-no-workaround *workaround_id[, workaround_id …]*

The `-no-workaround` (disable avoidance of specific errata) switch disables compiler code generator workarounds for specific hardware errata. See Controlling Silicon Revision and Anomaly Workarounds Within the Compiler for details of valid workarounds and the interactions of the `-si-revision`, `-workaround`, and `-no-workaround` switches.

-normal-word-code

The `-normal-word-code` switch has the same effect as compiling with the `-nwc` switch. It directs the compiler to generate instructions of normal word size (48-bits). This switch applies only when compiling code targeted for processors that support VISA execution, shown in the ***Processors Supporting VISA Execution*** table (see Processor Features).

> **NOTE:** Invoke this switch in the IDE by setting ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor*** > ***Variable Instruction Set Encoding (VISA)*** to ***Generate Normal Word code***.

-nwc

The `-nwc` switch has the same effect as compiling with the `-normal-word-code` switch. It directs the compiler to generate instructions of normal word size (48-bits). This switch applies only when compiling code targeted for processors that support VISA execution, shown in the ***Processors Supporting VISA Execution*** table (see Processor Features).

NOTE: Invoke this switch in the IDE by setting ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor*** > ***Variable Instruction Set Encoding (VISA)*** to ***Generate Normal Word code***.

-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. (Note that the switch settings begin with the uppercase letter "O" and end with a digit - a zero or a one.) The switch setting `-O` or `-O1` turns optimization on, while setting `-O0` turns off all optimizations.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Enable optimization***.

-Oa

The `-Oa` (automatic function inlining) switch enables the inline expansion of C/C++ functions, which are not necessarily declared inline in the source code. The amount of auto-inlining the compiler performs is controlled using the `-Ov num` (optimize for speed versus size) switch. 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[0|1]`.

NOTE: Invoke this switch in the IDE by setting ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Inlining*** to ***Automatic***.

-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 informs the compiler of the relative importance of speed versus size, when considering whether such trade-offs are worthwhile. The *num* variable should be an integer between 0 (purely size) and 100 (purely speed).

For any given optimization, the compiler modifies the code being generated. Some optimizations produce code that will execute in fewer cycles, but which will require more code space. In such cases, there is a trade-off between speed and space.

The *num* variable indicates a sliding scale between 0 and 100 which is the probability that a linear piece of generated code-a "basic block"-will be optimized for speed or for space. At `-Ov0` all blocks are optimized for space (equivalent to `-Os`) and at `-Ov100` all blocks are optimized for speed (equivalent to `-O`). At any point in between, the decision is based upon *num* and how many times the block is expected to be executed-the "execution count" of the block. The ***-Ov Switch Optimization Curve*** figure demonstrates this relationship.

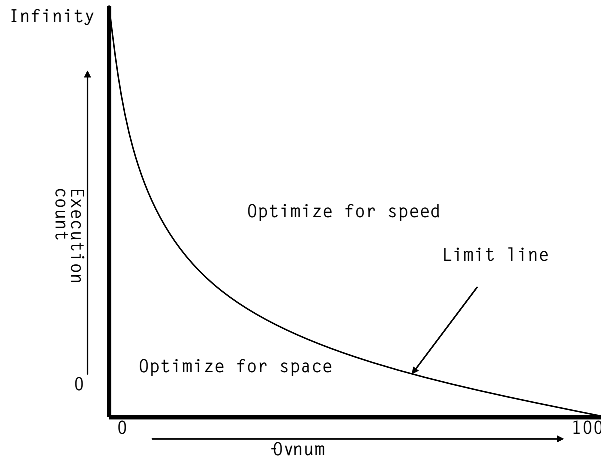

Figure 2-1: -Ov Switch Optimization Curve

For any given optimization where speed and size conflict, the potential benefit is dependent on the execution count: an optimization that increases performance at the expense of code size is considerably more beneficial if applied to the core loop of a critical algorithm than if applied to one-time initialization code or to rarely-used error-handling functions. If code appears to be executed only once, it will be optimized for space. As its execution count increases, so too does the likelihood that the compiler will consider the code increase worthwhile for the corresponding benefit in performance.

As the ***-Ov Switch Optimization Curve*** figure shows, the `-Ov` switch affects the point at which a given execution count is considered sufficient to switch optimization from "for space" to "for speed". Where *num* is a low value, the compiler is biased towards space, so a block's execution count has to be relatively high for the compiler to apply code-increasing transformations. Where *num* has a high value, the compiler is biased towards speed, so the same transformation will be considered valid for a much lower execution count.

The `-Ov` switch is most effective when used in conjunction with profile- guided optimization, where accurate execution counts are available. Without profile-guided optimization, the compiler makes estimates of the relative execution counts using heuristics.

> **NOTE:** Invoke this switch in the IDE by entering an appropriate value into the ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Optimize for code size/speed*** field.

For more information, see Using Profile-Guided Optimization.

-o *filename*

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

-overlay

The `-overlay` (program may use overlays) switch will disable the propagation of register information between functions and force the compiler to assume that all functions clobber all scratch registers. Note that this switch will affect all functions in the source file, and may result in a performance degradation. For information on disabling the propagation of register information only for specific functions, see #pragma overlay.

-overlay-clobbers *clobbered-regs*

The `-overlay-clobbers` (registers clobbered by overlay manager) switch identifies the set of registers clobbered by an overlay manager, if one is used. The compiler will assume that any call to an overlay-managed function will clobber the values in `clobbered-regs`, in addition to those clobbered by the function in question. A function is considered to be an overlay-managed function if the `-overlay` switch is specified, or if the function is marked with #pragma overlay.

The *`clobbered-regs`* argument is a single string, formatted as per the argument to `#pragma regs_clobbered`, except that individual components of the list may also be separated by commas.

> **NOTE:** Whitespace and semicolons are valid separators for the components of the list, but must be properly quoted when being passed to the compiler.

Examples:

```
cc21k -O t.c -overlay -overlay-clobbers r3,m4,r5
cc21k -O t.c -overlay -overlay-clobbers Dscratch
cc21k -O t.c -overlay -overlay-clobbers "r3 m4;r5"
```

-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.

-p

The `-p` (generate instrumented profiling) switch directs the compiler to generate the additional instructions needed to profile the program by recording the number of cycles spent in each function.

Applications built with the `-p` switch write the data to a `.prf` file. For more information on profiling, see Profiling With Instrumented Code.

> **NOTE:** Invoke this switch in the IDE via ***Project >Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Profiling*** > ***Enable compiler instrumented profiling***.

-path-{ asm | compiler | ipa | lib | link | prelink } *pathname*

The `-path-{asm|compiler|ipa|lib|link|prelink}` *`pathname`* (tool location) switch directs the compiler to use the specified component in place of the default-installed version of the compilation tool. The component comprises a relative or absolute path to its location. Respectively, the tools are the assembler, compiler, IPA solver, library builder, linker or prelinker. 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.

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

-pgo-session *session-id*

The -pgo-session (specify PGO session identifier) switch is used with profile-guided optimization. It has the following effects:

- When used with the -pguide switch, the compiler associates all counters for this module with the session identifier session-id.
- When used with a previously-gathered profile (a .pgo file), the compiler ignores the profile contents, unless they have the same session-id identifier.

This is most useful when the same source file is being built in more than one way (for example, different macro definitions, or for multiple processors) in the same application; each variant of the build can have a different session-id associated with it, which means that the compiler will be able to identify which parts of the gathered profile should be used when optimizing for the final build.

If each source file is only built in a single manner within the system (the usual case), then the -pgo-session switch is not needed.

> **NOTE:** Invoke this switch in the IDE by entering a suitable name into the ***Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization > PGO Session name*** field.

For more information, see for more information see, Using Profile-Guided Optimization.

-pguide

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

> **NOTE:** Invoke this switch in the IDE via ***Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Profile-guided Optimization > Prepare application to create new profile.***

For more information, see Using 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 shown in the ***Key Characters*** table.

Table 2-15: Key Characters

Character	*Meaning*
N	Normal line of source
X	Expanded line of source
S	Line of source skipped by #if or #ifdef
L	Change in source position
R	Diagnostic message (remark)
W	Diagnostic message (warning)
E	Diagnostic message (error)
C	Diagnostic message (catastrophic error)

-proc *processor*

The -proc (target processor) switch specifies the compiler-produced code suitable for the specified processor. Refer to CCES online help for the list of supported SHARC processors. For example,

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

NOTE: If no target is specified with the -proc switch, the system uses the ADSP-21160 processor settings as a default.

When compiling with the -proc switch, a hierarchy of processor macros are defined as 1 to identify the processor being used and the families it belongs to. So in the above example, __ADSP21161__, __ADSP21161_FAMILY__, __ADSP2116x__, __ADSP211xx__, __SIMDSHARC__, and __ADSP21000__ are all 1. __ADSPSHARC__ is defined to 0x110. For more information, see Predefined Preprocessor Macros.

See also -si-revision *version* for more information on silicon revision of the specified processor.

-prof-hw

The -prof-hw switch instructs the compiler to generate profiling code that shall be run on hardware (rather than on the simulator). The switch requires a supported profiling switch to also be specified on the command line. Supported profiling methods are profile-guided optimization (-pguide).

NOTE: Instrumented profiling (-p) does not differentiate between execution on hardware or simulator and can be executed on both targets. It does not require the -prof-hw switch.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Profile-guided Optimization*** > ***Gather profile using hardware***.

-progress-rep-func

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

-progress-rep-opt

The `-progress-rep-opt` switch provides feedback on the compiler's progress that may be useful when compiling and optimizing a very large, complex function. It issues a "warning" message each time the compiler starts a new optimization pass on the current function. The "warning" message is a remark that is disabled by default, and this switch enables the remark as a warning. The switch is equivalent to `-Wwarn=cc1473`.

-progress-rep-timeout

The `-progress-rep-timeout` switch issues a diagnostic message if the compiler exceeds a time limit during compilation. This indicates the compiler is still operating, just taking a long time.

-progress-rep-timeout-secs ***secs***

The `-progress-rep-timeout-secs` switch specifies how many seconds must elapse during a compilation before the compiler issues a diagnostic message about the length of time the compilation has used so far.

-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`* switch is position-dependent on the command line. That is, it affects only source files that follow the option.

NOTE: 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.

NOTE: 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`, `ustat2`, `ustat3` and `ustat4`. 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.

When compiling with the `-restrict-hardware-loops maximum` switch, the compiler will assume that any functions called by the code being compiler also do not use more hardware loops than the number specified. It is therefore necessary to make sure you compile any called functions with this switch too. Functions in the standard libraries shipped with CCES do not use more than 3 levels of nested hardware loops.

C++ code makes implicit calls to code in the CCES libraries, to provide functionality such as exception handling and memory management. These functions may use a maximum of one hardware loop level. When compiling C++ code, it is therefore not possible to restrict the hardware loop usage to zero, since one loop level is required for these implicit calls. Use of the `-restrict-hardware-loops 0` switch to compile C++ code will result in a build-time error.

`-rtcheck`

The `-rtcheck` (run-time checking) switch directs the compiler to generate additional code to check at runtime for common programming errors. This switch is equivalent to specifying all of the following switches:

- `-rtcheck-arr-bnd`
- `-rtcheck-div-zero`
- `-rtcheck-heap`
- `-rtcheck-null-ptr`
- `-rtcheck-shift-check`
- `-rtcheck-stack`
- `-rtcheck-unassigned`

NOTE: Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor*** > ***Enable run-time checking***.

`-rtcheck-arr-bnd`

The `-rtcheck-arr-bnd` (check array bounds at runtime) switch directs the compiler to generate additional code to verify that array accesses are within the bounds of the array.

NOTE: Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

NOTE: Invoke this switch in the IDE via the run-time checking options under *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor*.

See also `-rtcheck`.

`-rtcheck-div-zero`

The `-rtcheck-div-zero` (check for division by zero at runtime) switch directs the compiler to generate additional code to verify that divisors are non-zero before performing division operations.

NOTE: Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

NOTE: Invoke this switch in the IDE via the run-time checking options under *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor*.

See also `-rtcheck`.

`-rtcheck-heap`

The `-rtcheck-heap` (check heap operations at runtime) switch directs the compiler to link against the debugging variant of the heap library. For more information, see Heap Debugging.

NOTE: Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

NOTE: Invoke this switch in the IDE via the run-time checking options under *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor*.

See also `-rtcheck`.

`-rtcheck-null-ptr`

The `-rtcheck-null-ptr` (check for NULL pointers at runtime) switch directs the compiler to generate additional code to verify that pointers are not NULL, before dereferencing them.

NOTE: Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

NOTE: Invoke this switch in the IDE via the run-time checking options under *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Processor*.

See also `-rtcheck`.

-rtcheck-shift-check

The `-rtcheck-shift-check` (check shift values at runtime) switch directs the compiler to generate additional code to verify that, when shifting a value X by some amount Y, Y is a positive amount, and less than the number of bits used to represent X's type.

NOTE: Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

NOTE: Invoke this switch in the IDE via the run-time checking options under ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor***.

See also `-rtcheck`.

-rtcheck-stack

The `-rtcheck-stack` (check for stack overflow at runtime) switch directs the compiler to link with modified startup code and interrupt vector code that uses the CB7I interrupt to trap occurrences of stack overflow. For more information, see Stack Overflow Detection.

NOTE: Invoke this switch in the IDE via the run-time checking options under ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor***.

See also `-rtcheck`.

-rtcheck-unassigned

The `-rtcheck-unassigned` (check variables are assigned at runtime) switch directs the compiler to generate additional code to verify that variables have been assigned a value before they are used.

NOTE: Because of the additional overhead imposed by the checking code, this switch should only be employed during application development, and should not be used to build products for release.

NOTE: Invoke this switch in the IDE via the run-time checking options under ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor***.

See also `-rtcheck`.

-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.

-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.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Linker*** > ***General*** > ***Strip all symbols***.

-sat-associative

The `-sat-associative` (saturating addition is associative) switch instructs the compiler to consider saturating addition operations as associative: `(a+b)+c` may be rewritten as `a+(b+c)`, when the addition operator saturates. The default is that saturating addition is not associative.

-save-temps

The `-save-temps` (save intermediate files) switch directs the compiler to retain intermediate files, generated and normally removed as part of the various compilation stages. These intermediate files are placed in the `-path-output` specified output directory or the build directory if the `-path-output` switch is not used. See the ***C/C++ Compiler Common Switches*** table in Compiler Command-Line Switches for a list of intermediate files.

> **NOTE:** Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Save temporary 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_name` section as specified on the command line.

The compiler currently supports the following section identifiers; see Placement of Compiler-Generated Code and Data for more information.

`code`	Controls placement of machine instructions. Default is either `seg_pmco` or `seg_swco`, depending on whether normal-word code or short-word code is being generated.
`data`	Controls placement of initialized variable data. Default is `seg_dmda`.
`pm_data`	Controls placement of initialized data declared with the `_pm` keyword.
`constdata`	Controls placement of constant data.
`pm_constdata`	Controls placement of constant data declared with the `_pm` keyword.
`bsz`	Controls placement of zero-initialized variable data. Default is `seg_dmda`.
`sti`	Controls placement of the static C++ class constructor "start" functions. Default is `seg_pmco`. For more information, see Constructors and Destructors of Global Class Instances.
`switch`	Controls placement of jump-tables used to implement C/C++ switch statements.
`strings`	Controls placement of string literals.
`vtbl`	Controls placement of the C++ virtual lookup tables. Default is `seg_vtbl`.
`vtable`	Synonym for `vtbl`.
`autoinit`	Controls placement of data used to initialise aggregate autos.
`alldata`	Controls placement of `data`, `constdata`, `bss`, `strings` and `autoinit` all at once.

Note that `alldata` is not a real section *`kind`*, but rather a placeholder for `data`, `constdata`, `bsz`, `strings` and `autoinit`. Therefore,

```
-section alldata=X
```

is equivalent to

```
-section data=X
-section constdata=X
-section bsz=X
-section strings=X
-section autoinit=X
```

Ensure that the section selected via the command line exists within the `.ldf` file. Refer to the ***Linker*** chapter in the *Linker and Utilities Manual*.

-short-word-code

The `-short-word-code` switch has the same effect as compiling with the `-swc` switch. It directs the compiler to generate instructions of short word size (16/32/48-bits). This switch only applies when compiling code targeted for processors that support VISA execution, shown in the ***Processors Supporting VISA Execution*** table (see Processor Features).

NOTE: Invoke this switch in the IDE by setting ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor*** > ***Variable Instruction Set Encoding (VISA)*** to ***Generate VISA code***.

-show

The `-show` (display command line) switch shows the command-line arguments passed to `cc21k`, including expanded option files and environment variables. This option allows you to ensure that command-line options have been passed successfully.

-si-revision *version*

The `-si-revision` *`version`* (silicon revision) switch directs the compiler to build for a specific hardware revision (version). Any errata workarounds available for the targeted silicon revision will be enabled. For more information, see Controlling Silicon Revision and Anomaly Workarounds Within the Compiler.

-signed-bitfield

The `-signed-bitfield` (make plain bit-fields signed) switch directs the compiler to make plain bit-fields -- those which have not been declared with an explicit signed or unsigned keyword -- be signed. This is the default mode. For more information, see `-unsigned-bitfield`.

-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.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Language Settings*** > ***Structs/classes do not overlap***.

-swc

The -swc switch has the same effect as compiling with the -short-word-code switch. It directs the compiler to generate instructions of short-word size (16/32/48-bits). This switch only applies when compiling code targeted for processors that support VISA execution, shown in the ***Processors Supporting VISA Execution*** table (in Processor Features).

NOTE: Invoke this switch in the IDE by setting ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor*** > ***Variable Instruction Set Encoding (VISA)*** to ***Generate VISA code***.

-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.

The ***System Macros Defined*** table shows the macros if the system returns information.

Table 2-16: System Macros Defined

Macro	*Description*
__HOSTNAME__	The name of the host machine
__SYSTEM__	The Operating System name of the host machine
__USERNAME__	The current user's login name

-T *filename*

The -T (linker description file) switch directs the compiler to use the specified linker description file (.ldf) as control input for linking. If -T is not specified, a default .ldf file is selected based on the processor variant.

-threads

The -threads switch directs the compiler to link against the thread-safe variants of the C/C++ run-time libraries. When used, the -threads switch defines the macro _ADI_THREADS as one (1) at the compile, assemble and link phases of a build.

See also -no-threads.

NOTE: The use of the `-threads` switch does not imply that the compiler will produce thread-safe code when compiling C/C++ source. Make sure to use multi-threaded programming practices in your code.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Linker*** > ***Processor*** > ***Link against thread-safe run-time libraries***.

-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.

-U *macro*

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.

See also `-D` *macro*[=*definition*].

NOTE: Add instances of this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Preprocessor*** > ***Preprocessors undefines***.

-unsigned-bitfield

The `-unsigned-bitfield` (make plain bit-fields unsigned) switch directs the compiler to make plain bit-fields-those which have not been declared with an explicit `signed` or `unsigned` keyword - be unsigned.

For example, given the declaration

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

The ***Bit-Field Values*** table lists the `bit-field` values.

Table 2-17: Bit-Field Values

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

See also `-signed-bitfield`.

-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 its version information.

-W{annotation|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 (annotations, 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 you can override the severity of the diagnostic. Each diagnostic message is identified by a number that is the same in all compiler software releases.

> **NOTE:** If the processing of the compiler command line generates a diagnostic, the position of the `-W` switch on the command line is important. If the `-W` switch changes the severity of the diagnostic, it must occur before the command line switch that generates the diagnostic; otherwise, no change of severity will occur.
>
> Also, as shown in the ***Console*** view and in help, error codes sometimes begin with a leading zero (for example, `cc0025`). If you try to suppress error codes with `#pragma diag()` or `-W{annotation|error|remark|suppress|warn}` *`number[, number ...]`*, and supply the code with a leading zero, it does not work. This is because the compiler reads the number as an octal value, and will suppress a different warning or error.

-Wannotations

The `-Wannotations` (enable code generation annotations) switch directs the compiler to issue code generation annotations, which are messages milder than warnings that may help you to optimize your code.

> **NOTE:** Invoke this switch in the IDE by settings ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Warning*** > ***Warning/annotation/remark control*** to ***Errors, warnings and annotations***.

-Werror-limit ***number***

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

-Werror-warnings

The `-Werror-warnings` (treat warnings as errors) switch directs the compiler to treat all warnings as errors, with the result that a warning will cause the compilation to fail.

-Wremarks

The -Wremarks (enable diagnostic warnings) switch directs the compiler to issue remarks, which are diagnostic messages milder than warnings. Code generation annotations also is issued, unless disabled with the -no-annotate switch.

NOTE: Invoke this switch in the IDE by settings ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Warning*** > ***Warning/annotation/remark control*** to ***Errors, warnings, annotations and remarks***.

-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.

NOTE: Invoke this switch in the IDE by settings ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Warning*** > ***Warning/annotation/remark control*** to ***Errors only***.

NOTE: If the processing of the compiler command line generates a warning, the position of the -w switch on the command line is important. If the -w switch is located before the command line switch that causes the warning, the warning will be suppressed; otherwise, it will not be suppressed.

-warn-component

The -warn-component (warn if component elements are missing) switch instructs the compiler to issue warnings if it cannot locate libraries that are requested by the XML file of the component. For more information, see -component *file.xml* .

-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.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Warning*** > ***Function declarations without prototypes***.

-workaround *workaround_id[, workaround_id…]*

The -workaround (enable avoidance of specific errata) switch enables compiler code generator workarounds for specific hardware errata.

See Controlling Silicon Revision and Anomaly Workarounds Within the Compiler for details of valid workarounds and the interaction of the -si-revision, -workaround, and -no-workaround switches.

-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 one of the characters listed in the ***ref-code Characters*** table.

Table 2-18: ref-code Characters

Character	*Meaning*
D	Definition
d	Declaration
M	Modification
A	Address taken
U	Used
C	Changed (used and modified)
R	Any other type of reference
E	Error (unknown type of reference)

NOTE: Please note that the compiler's `-xref` switch differs from the `-xref` switch used by the linker. Refer to the *Linker and Utilities Manual* for more information.

C Mode (MISRA) Compiler Switch Descriptions

The following switches apply only to the C compiler. See MISRA-C Compiler for more information.

`-misra`

The `-misra` switch enables checking for MISRA-C Guidelines. Some rules or parts of rules are relaxed with this switch enabled. Rules relaxed by this option are 5.1, 5.7, 6.3, 6.4, 8.1, 8.2, 8.5, 10.3, 10.4, 10.5, 12.8, 13.7 and 19.7. This is explained in more detail, see Rules Descriptions.

The `-misra` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions. The switch predefines the `_MISRA_RULES` preprocessor macro.

`-misra-linkdir` ***`directory`***

The `-misra-linkdir` switch specifies a directory in which to place `.misra` files. The default is a local directory called `MISRARepository`. The `.misra` files enable checking of violations of rules 5.5, 8.8 and 8.10.

`-misra-no-cross-module`

The switch implies `-misra`, but also disables checking for a number of rules that require the use of the prelinker to check across multiple modules for rule violation. The MISRA-C rules suppressed are 5.5, 8.8 and 8.10.

The `-misra-no-cross-module` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions.

-misra-no-runtime

The switch implies `-misra`, but also disables run-time checking for MISRA-C rules 21, 17.1, 17.2 and 17.3. It limits the checking of rules 9.1, 12.8, 16.2 and 17.4.

The `-misra-no-runtime` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions.

-misra-strict

The `-misra-strict` switch enables checking for MISRA-C Guidelines. The switch ensures a strict interpretation of the MISRA-C: 2004 Guidelines. See Rules Descriptions for details.

The `-misra-strict` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions. The switch predefines the `_MISRA_RULES` preprocessor macro.

-misra-suppress-advisory

The switch implies `-misra`, but suppresses the reporting of advisory rules.

The `-misra-suppress-advisory` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions.

-misra-testing

The switch implies `-misra`, but also suppresses checking of MISRA-C rules 20.4, 20.7, 20.8, 20.9, 20.10, 20.11, and 20.12.

The `-misra-testing` switch is not supported in conjunction with some switches. For more information, see MISRA-C Command-Line Switch Restrictions.

-Wmis_suppress *rule_number[, rule_number]*

The `-Wmis_suppress` switch with a *rule_number* argument directs the compiler to suppress the specified diagnostic for a MISRA-C rule. The *rule_number* argument identifies the specific message to override.

-Wmis_warn *rule_number[, rule_number]*

The `-Wmis_warn` switch with a *rule_number* argument directs the compiler to override the severity of the specified diagnostic to produce a warning for a MISRA-C rule. The *rule_number* argument identifies the specific message to override.

MISRA-C Command-Line Switch Restrictions

The ***Switches Disallowed by MISRA-C*** table lists the command-line switches that are disallowed in MISRA-C mode.

Table 2-19: Switches Disallowed by MISRA-C

Switch Name
-w
-Wsuppress (-W{annotation|error|remark|suppress|warn} *number[, number ...]*)
-Wwarn (-W{annotation|error|remark|suppress|warn} *number[, number ...]*)

Table 2-19: Switches Disallowed by MISRA-C (Continued)

Switch Name
-c++
-c++11
-enum-is-int
-warn-protos
-alttok

C++ Mode Compiler Switch Descriptions

The following switches apply only when compiling in C++ mode.

-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. Use the -no-anach switch for greater standard compliance.

The following anachronisms are accepted in C++ mode when the -anach switch is enabled:

- 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++() function can be used to overload both prefix and postfix ++ operations.
- A single operator--() function can be used to overload both prefix and postfix -- operations.
- The base class name may be omitted in a base class initializer if there is only one immediate base class.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as an un-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-const type may be initialized from a value of a different type. A temporary is created; it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-const class type may be initialized from an rvalue of the class type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following statements declare the overload of two functions named f.

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

-check-init-order

It is not guaranteed that global objects requiring constructors are initialized before their first use in a program consisting of separately compiled units. The compiler will output warnings if these objects are external to the compilation unit and are used in dynamic initialization or in constructors of other objects. These warnings are not dependent on the `-check-init-order` switch.

In order to catch uses of these objects and to allow the opportunity for code to be rewritten, the `-check-init-order` (check initialization order) switch adds run-time checking to the code. This generates output to `stderr` that indicates uses of such objects are unsafe.

NOTE: This switch generates extra code to aid development, and should not be used when building production systems.

NOTE: Invoke this switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Language Settings*** > ***Check initialization order***.

-friend-injection

The `-friend-injection` switch directs the compiler to conform to perform name lookup in a non-standard way with respect to friend declarations. With this switch enabled a friend declaration will be injected into the scope enclosing the class containing the friend declaration.

See also `-no-friend-injection`.

-full-dependency-inclusion

The `-full-dependency-inclusion` switch ensures that when generating dependency information for implicitly-included `.cpp` files, the `.cpp` file will be re-included. This file is re-included only if the `.cpp` files are included more than once in the source (via re-inclusion of their corresponding header file). This switch is required only if your C++ sources files are compiled more than once with different macro guards.

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

-implicit-inclusion

The `-implicit-inclusion` switch directs the compiler to enable the implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. The compiler will automatically include a source file `.C`, `.c` or `.cpp` when the corresponding header file `.h` or `.hxx` is included. This is a mechanism that was common practice, but is not standard behavior.

See also `-no-implicit-inclusion` .

NOTE: This switch is incompatible with the use of exported templates.

-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 -anach for more information about these features. This is the default mode.

-no-friend-injection

The -no-friend-injection switch directs the compiler to conform to the ISO/IEC 14882:2003 standard with respect to friend declarations. The friend declaration is visible when the class of which it is a friend is among the associated classes considered by argument-dependent lookup. This is the default mode.

See also -friend-injection.

-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. This is compatible with the use of exported templates as defined by the ISO/IEC:14883 standard. This is the default mode. This switch is accepted but ignored when compiling C files.

See also -implicit-inclusion.

-no-rtti

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

See also -rtti.

-no-std-templates

The -no-std-templates switch disables dependent name processing, i.e, the special lookup of names used in templates as required by the C++ standard.

See also -std-templates .

-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 -no-rtti.

Invoke this switch in the IDE via *Project* > *Properties* > *C/C++ Build* > *Settings* > *Tool Settings* > *Compiler* > *Language Settings* > *C++ exceptions and RTTI.*

-std-templates

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

See also -no-std-templates.

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 CCES 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 prepended to the command line. Multiple switches are separated by spaces or new lines. A vertical-bar (|) character may be used to indicate that any switches following it will be processed after all other command-line switches.
- *CC21K_IGNORE_ENV* If this environment variable is set, `CC21K_OPTIONS` is ignored.

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. The following tables show the sizes used for each intrinsic C/C++ data type.

Table 2-20: Data Type Sizes for ADSP-21xxx and ADSP-SCxxx Processors (with no support for byte-addressing or when using -char-size-32)

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
`long long`	64 bits signed	2

Table 2-20: Data Type Sizes for ADSP-21xxx and ADSP-SCxxx Processors (with no support for byte-addressing or when using -char-size-32) (Continued)

Type	*Bit Size*	*Result of sizeof Operator*
`unsigned long long`	64 bits unsigned	2
`bool`	32 bits signed	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
`short fract`	32 bits fixed-point	1
`fract`	32 bits fixed-point	1
`long fract`	32 bits fixed-point	1
`unsigned short fract`	32 bits unsigned fixed-point	1
`unsigned fract`	32 bits unsigned fixed-point	1
`unsigned long fract`	32 bits unsigned fixed-point	1
`double`	either 32 or 64 bits float (default 32)	either 1 or 2 (default 1)
`long double`	64 bits float	2

Table 2-21: Data Type Sizes for ADSP-21xxx and ADSP-SCxxx Processors (with support for byte-addressing, when using -char-size-8)

Type	*Bit Size*	*Result of sizeof Operator*
`int`	32 bits signed	4
`unsigned int`	32 bits unsigned	4
`long`	32 bits signed	4
`unsigned long`	32 bits unsigned	4
`long long`	64 bits signed	8
`unsigned long long`	64 bits unsigned	8
bool	8 bits signed	1
`char`	8 bits signed	1
`unsigned char`	8 bits unsigned	1
`short`	16 bits signed	2
`unsigned short`	16 bits unsigned	2

Table 2-21: Data Type Sizes for ADSP-21xxx and ADSP-SCxxx Processors (with support for byte-addressing, when using -char-size-8) (Continued)

Type	*Bit Size*	*Result of sizeof Operator*
`pointer`	32 bits	4
`float`	32 bits float	4
`short fract`	32 bits fixed-point	4
`fract`	32 bits fixed-point	4
`long fract`	32 bits fixed-point	4
`unsigned short fract`	32 bits unsigned fixed-point	4
`unsigned fract`	32 bits unsigned fixed-point	4
`unsigned long fract`	32 bits unsigned fixed-point	4
`double`	either 32 or 64 bits float (default 32)	either 4 or 8 (default 4)
`long double`	64 bits float	8

The Analog Devices compiler does not support data sizes smaller than the addressable unit size on the processor. For ADSP-21xxx and ADSP-SCxxx processors without byte-addressing support (see the ***Processors Supporting Byte-Addressing*** table in Processor Features), 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 how to use the fixed-point data types in C, refer to Using Native Fixed-Point Types.

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. 64-bit integer support is provided by the `long long` and `unsigned long long` data types, which are emulated data types, implemented through software.

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, computations using double word data types run more slowly than with 32-bit data types. On processors without native hardware support for double-precision floating-point arithmetic (see the ***Processors Supporting Native Double-Precision Floating-Point Arithmetic*** table in Processor Features), such computation relies largely on software-emulated arithmetic. On processors with native double-precision floating-point support, the instructions take longer to execute than single-precision floating-point instructions.

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, which 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 ISO/IEC

9899: 1990 C standard, the ISO/IEC 9899: 1999 C standard or the ISO/IEC 14882: 2003 C++ standard, which all require that the -double-size-64 switch be enabled.

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 */
```

Full descriptions of these functions and their implementation can be found in the ***C/C++ Library Manual for SHARC Processors***.

Optimization Control

The general aim of compiler optimization is to generate correct code that executes quickly and is small in size. Not all optimizations are suitable for every application or 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 or pragmas.

NOTE: Refer to the Optimal Performance from C/C++ Source Code chapter for information on how to obtain maximal code performance from the compiler.

Optimization Levels

The following list identifies several optimization levels. The levels are notionally ordered with least optimization listed first and most optimization listed last. The descriptions for each level outline the optimizations performed by the compiler and identify any 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 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 CCES IDE). Default optimization level can be enabled using the optimize_off pragma. See General Optimization Pragmas.

- *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], -Os, and -Ov *num*. Procedural optimizations for speed and space (-O and -Os) can be enabled in C/C++ source using the pragma optimize_{for_speed|for_space}.

For more information, see General Optimization Pragmas.

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

  The compiler performs advanced aggressive optimizations using profiler statistics (.pgo files) generated from running the application using representative training data. PGO can be used in conjunction with IPA and automatic inlining. See -pguide for more information.

  The most common scenario in collecting PGO data is to set up one or more simple file-to-device streams where the file is a standard ASCII stream input file and the device is any stream device supported by the simulator target, such as memory and peripherals. The PGO process can be broken down into the execution of one or more data sets where a data set is the association of zero or more input streams with one and only one .pgo output file.

  For more information, see Using Profile-Guided Optimization.

  NOTE: Be aware of the requirement for allowing command-line arguments in your project when using PGO. See Support for argv/argc for details.

- ***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 which additionally enables procedural optimizations (-O). See -Oa , -Ov *num*, -O[0|1], and Function Inlining for more information.

  NOTE: 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 which additionally enables procedural optimizations (-O). See Interprocedural Analysis, -ipa, and -O[0|1] 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 for more information.

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 the Optimal Performance from C/C++ Source Code chapter.

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.

Enable interprocedural analysis by selecting the ***Interprocedural analysis*** check box on the ***Compile*** > ***General*** page of the CCES ***Properties*** dialog box, or by specifying the `-ipa` command-line switch.

The `-ipa` switch automatically enables the `-O` switch to turn on optimization. See also `-ipa`.

Use of the `-ipa` switch generates additional files along with the object file produced by the compiler. These files have `.ipa` 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, recompiling source files where necessary, to make use of gathered information.

NOTE: Because a file may be recompiled by the prelinker, do not use the `-S` option to see the final optimized assembler file when `-ipa` is enabled. Instead, use the `-save-temps` switch, so that the full compile/link cycle can be performed first.

Interaction With Libraries

When IPA is enabled, the compiler examines all of the source files to build 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 by recompiling where necessary.

Because IPA operates only during the final link, the `-ipa` switch has no benefit when initially compiling source files to object format for inclusion in a library. IPA gathers information about each file and embeds this within the object format, but cannot make use of it at this point, because the library contents have not yet been used in a specific context.

When IPA is invoked during linking, it will recover the gathered information from all linked-in object files that were built with `-ipa`, and where necessary and possible, will recompile source files to apply additional optimizations. Modules linked in from a library are not recompiled in this manner, as source is not available for them. Therefore, the gathered information in a library module can be used to further optimize application sources, but does not provide a benefit to the library module itself.

If a library module makes references to a function in a user module in the program, this will be detected during the initial linking phase, and IPA will not eliminate the function. If the library module was not compiled with `-ipa`,

IPA will not make any assumptions about how the function may be called, so the function may not be optimized as effectively as if all references to it were in source code visible to IPA, or from library modules compiled with `-ipa`.

Controlling Silicon Revision and Anomaly Workarounds Within the Compiler

The compiler provides three switches which specify that code produced by the compiler will be generated for a specific revision of a specific processor, and appropriate silicon revision targeted system run time libraries will be linked against. Targeting a specific processor allows the compiler to produce code that avoids specific hardware errata reported against that revision. For the simplest control, use the `-si-revision` switch which automatically controls compiler workarounds.

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

When developing using the CCES IDE, the silicon revision used to build sources is part of a projects processor settings.

This section describes:

- Using the `-si-revision` Switch
- Using the `-workaround` Switch
- Using the `-no-workaround` Switch
- Interactions Between the Silicon Revision and Workaround Switches
- Anomalies in Assembly Sources

Using the `-si-revision` Switch

The `-si-revision` *`version`* (silicon revision) switch directs the compiler to build for a specific hardware revision. Any errata workarounds available for the targeted silicon revision will be enabled. The parameter *`version`* represents a silicon revision of the processor specified by the `-proc` *`processor`* switch.

For example,

```
cc21k -proc ADSP-21161 -si-revision 0.1 prog.c
```

If silicon version `any` is used, then errata workarounds are enabled for all supported revisions of 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 at the time of release, and any errata workarounds which are appropriate for the latest silicon revision will be enabled.

In the `SHARC\lib` CCES installation directory there are a number of subdirectories. Within each of these is a complete set of libraries built for specific parts and silicon revisions. When linking an executable, the compiler driver selects and links against the best of these sets of libraries that is correct for the target part and has been built with the necessary silicon anomaly workarounds enabled to match the silicon revision switch. Note that an individual set of libraries may cover more than one specific part or 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 a hexadecimal number 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.

The compiler driver will pass the -si-revision switch, as specified in the command line, when invoking other tools in the CCES toolchain.

NOTE: Visit www.analog.com/processors/tools/anomalies to get more information on specific anomalies, including anomaly IDs.

Using the -workaround Switch

The -workaround *workaround_id[, workaround_id…]* switch enables compiler code generator workarounds for specific hardware errata.

When workarounds are enabled, the compiler defines the macro __WORKAROUNDS_ENABLED at the compile, assembly, and link build stages. The compiler also defines individual macros for each of the enabled workarounds for each of these stages, as indicated by each macro description.

For a complete list of anomaly workarounds and associated *workaround_id* keywords, refer to the anomaly .xml files provided in the <install_path>\System\ArchDef directory. These are named in the format <platform_name>-anomaly.xml.

To find which workarounds are enabled for each chip and silicon revision, refer to the appropriate <*chip_name*>-compiler.xml file in the same directory (for example, ADSP-21488-compiler.xml). Each *-compiler.xml file references an *-anomaly.xml file via the name in the <cces-anomaly-dictionary> element.

The anomaly .xml files relevant to SHARC processors have filenames of the form SHARC-21xxx-anomaly.xml or SHARC-SCxxx-anomaly.xml.

Using the -no-workaround Switch

The -no-workaround *workaround_id[, workaround_id …]* switch disables compiler code generator workarounds for specific hardware errata. For a complete list of valid workaroundID values, refer to the relevant *-anomaly.xml file. For more information, see Using the -workaround Switch.

The -no-workaround switch can be used to disable workarounds enabled via the -si-revision *version* or -workaround *workaroundID* switch.

All workarounds can be disabled by providing -no-workaround with all valid workarounds for the selected silicon revision or by using the option -no-workaround all. Disabling all workarounds via the -no-workaround switch will link against libraries with no silicon revision in cases where the silicon revision is not none.

Interactions Between the Silicon Revision and Workaround Switches

The interactions between `-si-revision`, `-workaround`, and `-no-workaround` can only be determined once all the command line arguments have been parsed.

To this effect options will be evaluated as follows:

1. The `-si-revision` *`version`* is parsed to determine which revision of the run-time libraries the application will link against. It also produces an initial list of all the default compiler errata workarounds to enable.
2. Any additional workarounds specified with the `-workaround` switch will be added to the errata list.
3. Any workarounds specified with `-no-workaround` will then be removed from this list.
4. If any workarounds were declared via `-workaround`, the macro `__WORKAROUNDS_ENABLED` is defined at compile and assembly and link stages, even if `-no-workaround` disables all workarounds.

Anomalies in Assembly Sources

If your project includes some hand-written assembly code, you will have to ensure that you explicitly avoid any relevant anomalies that apply to your target processor. This can be simplified by the use of the `sys\anomaly_macros_rtl.h` header file. This header file defines macros for each of the anomalies that affect the run-time libraries, which allow for conditional inclusion of avoidance code.

For example, the following code makes use of the `WA_09000014` macro to conditionally select code that avoids problems with conditional stores in delay slots.

```
#if WA_09000014
   r4 = r4 - r12;
   if not ac dm(-3, i3) = r12;
   jump(pc, exit_malloc);
#else
   jump(pc, exit_malloc) (DB);
   r4 = r4 - r12;
   if not ac dm(-3, i3) = r12;
#endif
```

Using Byte-Addressing

The processors in the ***Processors Supporting Byte-Addressing*** table (see Processor Features) have the ability to access individual 8-bit bytes in memory. This feature is known as byte-addressing. Each individual 8-bit byte has a unique address, in contrast to earlier SHARC families which, in the C runtime environment, permitted access to a minimum of 32 bits at a time (known as word-addressing). Byte-addressing permits compatibility with code written assuming that the C/C++ `char` type is 8 bits in length, and the C/C++ `short` type is 16 bits in length.

This chapter discusses how to use byte-addressing with CrossCore Embedded Studio.

- Instructions on how to build an entirely byte-addressed application are given in Building Byte-Addressed Applications From C/C++.

- Differences between data formats and calling conventions for byte-addressed and word-addressed code are given in Data Formats and Calling Conventions.
- The use of `#pragma byte_addressed` and `#pragma word_addressed` to facilitate use of functions and data compiled for word-addressing within byte-addressed applications, and vice versa, is discussed in Mixed Char-Size Applications.
- Finer-grained control of char-size is discussed in Constructing Complex Byte-Addressed and Word-Addressed Interfaces.
- Extra considerations relating to the use of byte-addressed and word-addressed code together in C++ applications is discussed in Using Byte-Addressed and Word-Addressed Interfaces with C++.
- Using assembly code in mixed byte-addressed and word-addressed applications is discussed in Assembly Code.

Building Byte-Addressed Applications From C/C++

CrossCore Embedded Studio supports building applications both using word-addressing, enabling compatibility with existing SHARC applications, and using byte-addressing. Byte-addressing is enabled by the command line switch `-char-size-8`, which is the default setting, while word-addressing is enabled by the command line switch `-char-size-32`. Char-size may also be selected in the CrossCore Embedded Studio IDE through ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***CrossCore SHARC C/C++ Compiler*** > ***Processor*** > ***Char size***.

Data Formats and Calling Conventions

If you are used to working with word-addressed code on earlier generations of the SHARC processor and are moving to byte-addressed code, there are several differences of you should be aware. These are:

- The sizes of native integer types, discussed in Sizes of Native Types.
- The endianness of 64-bit scalar types and bitfields, discussed in Endianness.
- The scheme used for translating C/C++ source names into linkage names, discussed in Symbol Names.
- The address space used for accesses to memory, discussed in Byte-Addressed Memory Alias.
- Changes to the call-preserved register set, discussed in Changes to Call-Preserved Registers.

Sizes of Native Types

Sizes of the native C/C++ types when using the two switches are shown in the ***Sizes of C/C++ Native Types With -char-size-8 and -char-size-32*** table.

Table 2-22: Sizes of C/C++ Native Types With -char-size-8 and -char-size-32

Type	*Size in Bits* (`-char-size-8`)	*Size in Bits* (`-char-size-32`)
`char`	8	32
`short`	16	32

Table 2-22: Sizes of C/C++ Native Types With -char-size-8 and -char-size-32 (Continued)

Type	*Size in Bits* (**-char-size-8**)	*Size in Bits* (**-char-size-32**)
int	32	same as -char-size-8
long long	64	same as -char-size-8
float	32	same as -char-size-8
double (*)	32 or 64	same as -char-size-8
long double	64	same as -char-size-8
short fract	32	same as -char-size-8
fract	32	same as -char-size-8
long fract	32	same as -char-size-8
pointer (e.g. int *)	32	same as -char-size-8

NOTE: The signed and unsigned variants of these types are the same size as the corresponding plain type. The size of the double type depends on use of the switches -double-size-32 and -double-size-64.

Endianness

64-bit values of type long long, unsigned long long, long double, and double (when the -double-size-64 switch is used) are stored in memory in big-endian format when compiling with -char-size-32, but are stored in little-endian format when compiling with -char-size-8. This is in order to promote easy inter-operation of the SHARC core with other cores and devices that store values in little-endian format. An example of how 64-bit values are laid out in memory is shown in the ***Layout of 64-Bit Values in Memory When Building With -char-size-8 and -char-size-32*** table.

Table 2-23: Layout of 64-bit Values in Memory When Building With -char-size-8 and -char-size-32

Data	**-char-size-8** *Address: Value*	**-char-size-32** *Address: Value*
long long x = 0x1122334455667788LL;	0x2c0000: 0x88 0x2c0001: 0x77 0x2c0002: 0x66 0x2c0003: 0x55 0x2c0004: 0x44 0x2c0005: 0x33 0x2c0006: 0x22 0x2c0007: 0x11	0xb0000: 0x11223344 0xb0001: 0x55667788
long double y = 1.0;	0x2c0000: 0x00 0x2c0001: 0x00 0x2c0002: 0x00 0x2c0003: 0x00 0x2c0004: 0x00 0x2c0005: 0x00	0xb0000: 0x3ff00000 0xb0001: 0x00000000

Table 2-23: Layout of 64-bit Values in Memory When Building With -char-size-8 and -char-size-32 (Continued)

Data	**-char-size-8** *Address: Value*	**-char-size-32** *Address: Value*
	`0x2c0006: 0xf0` `0x2c0007: 0x3f`	

The endianness of bitfields is also affected by the char-size setting. When `-char-size-32` is used, the fields are laid out starting from the most-significant end of each word. When `-char-size-8` is used, the fields are laid out starting from the least-significant end of each word. This is shown in the ***Layout of Bitfields in Memory When Building With -char-size-8 and -char-size-32*** table.

Table 2-24: Layout of Bitfields in Memory When Building With -char-size-8 and -char-size-32

Data	**-char-size-8** *Address: Value*	**-char-size-32** *Address: Value*

`struct bits {`
`   int a:4;`
`   int b:8;`
`   int c:24;`
`} my_bits = {0x4, 0x8, 0x24};`
	`0x2c0000: 0x84`
	`0x2c0001: 0x00`
	`0x2c0002: 0x00`
	`0x2c0003: 0x00`
	`0x2c0004: 0x24`
	`0x2c0005: 0x00`
	`0x2c0006: 0x00`
	`0x2c0007: 0x00`
		`0xb0000: 0x40800000`
		`0xb0001: 0x00002400`

Symbol Names

As the sizes of native types and the pointer representation are different in word-addressed and byte-addressed code, word-addressed code must not used where byte-addressed code is expected or vice versa. This is ensured by using a different linkage name for code compiled with the `-char-size-8` and `-char-size-32` switches. With `-char-size-32`, an underscore is prefixed to the C identifier, while with `-char-size-8`, a period is appended to the C identifier. This is shown in the ***Linkage Names Used for C Symbols When Compiled With -char-size-8 and -char-size-32*** table.

Table 2-25: Linkage Names Used for C Symbols When Compiled With -char-size-8 and -char-size-32

C Name	**-char-size-8**	**-char-size-32**
`my_function`	`my_function.`	`_my_function`
`my_data`	`my_data.`	`_my_data`

Byte-Addressed Memory Alias

Unlike word-addresses, byte-addresses must be able to represent which byte within a word is being accessed, requiring a distinct byte-addressed address format. Memory on the ADSP-215xx and ADSP-SC5xx processors can be accessed using a byte-addressed address space, which uses this format. Most of the memory may additionally be accessed using a word-addressed alias (the normal-word, or NW, address space) as on the ADSP-211xx/212xx/213xx/

214xx SHARC processors, as well as the short-word (SW) and long-word (LW) address aliases. See the address map for more information.

In code compiled with `-char-size-8`, all pointers will by default be located in the byte-addressed alias space. This includes the frame and stack pointers `i6` and `i7`, and the addresses of any data buffers and global variables.

Changes to Call-Preserved Registers

When `-char-size-32` is used, the `MRF` and `MRB` registers are defined as call-preserved, so a function that modifies them is responsible for saving their values on entry and restoring these original values on exit from the function.

When `-char-size-8` is used, the `MRF` and `MRB` registers are defined as scratch registers. This means if they are modified in a function, their values are not restored on exit from the function. Therefore live values should not be kept in the `MRF` and `MRB` registers across a call to a function compiled with `-char-size-8`.

Mixed Char-Size Applications

This section contains:

- Building a Mixed Char-Size Application
- How to Use Different Char Sizes Together
- Type Coercions
- Simple Interfaces

Building a Mixed Char-Size Application

When building an entire application from C/C++ source, the `-char-size-8` and `-char-size-32` switches can be applied to the project as a whole. In this case, the sizes of native types, endianness, and mangling schemes are consistent across the entire application. In new applications, we recommended using the default setting (`-char-size-8`). The default setting offers greatest compatibility with code ported from other platforms and simplifies use of peripherals and inter-core communication.

In some cases, you may wish to reuse existing code written for earlier, word-addressed SHARC processors within an application using `-char-size-8`. Combining such code with the new `-char-size-8` parts of your application requires you to direct the compiler regarding the interface between the different application parts. The interface between the byte-addressed and word-addressed parts allows the compiler to perform any necessary conversions. This approach is discussed in How to Use Different Char Sizes Together.

To ensure that each source file in your project is built for the correct sizes of the native types, indicate the char-size setting. Indicate the setting for each source file whose char-size does not match the char-size set at the project options level. First, set the default char-size for the project using the ***Project Properties*** dialog box. Next, for each source file to built using the opposite char-size, set the ***Build Properties*** for that source file. The `char` size setting is found under ***C/C++ Build*** > ***Settings*** > ***Tools Settings*** > ***CrossCore SHARC C/C++ Compiler*** > ***Processor*** > ***Char size***.

How to Use Different Char Sizes Together

The differences in calling conventions and data layout discussed in Data Formats and Calling Conventions mean that the compiler needs to know the char-size setting which was used when compiling the source file where each function or variable is defined.

To do this, declare functions and data defined in source files using the opposite char-size using `#pragma byte_addressed` or `#pragma word_addressed`. `#pragma byte_addressed` is used to mark symbols found in source files built with `-char-size-8`, allowing them to be referenced from code built with either `-char-size-8` or `-char-size-32`. Likewise `#pragma word_addressed` is used to mark symbols found in source files built with `-char-size-32`.

These pragmas can be used in two ways. Firstly, the pragma can immediately precede a single function or external data declaration. For example, a source file built with `-char-size-8` might contain:

```
// function prototype for function built with -char-size-32
#pragma word_addressed
extern int my_char_size_32_func(void);

// function prototype for function built with same char-size as current
compilation
extern int my_char_size_8_func(void);
```

Alternatively the pragmas can be used with `push` and `pop` to enclose a region where all symbols and types belong to the opposite char-size. For example, we might declare the following in a translation unit built with `-char-size-8`:

```
#pragma word_addressed(push)
// function prototype for function built with -char-size-32
extern int my_char_size_32_func(void);

// function prototype for another function built with -char-size-32
extern void another_char_size_32_func(int a);
#pragma word_addressed(pop)

// function prototype for function built with same char-size as
// current compilation
int my_char_size_8_func(void);
```

Regardless of which method is used, we say such symbols are declared in an interface region.

Within an interface region of the opposite char-size, the compiler will interpret the declarations as they would be if compiled with that char-size. The sizes of basic types will be changed as per the ***Sizes of C/C++ Native Types With -char-size-8 and -char-size-32*** table (in Sizes of Native Types). Interface regions also tell the compiler about the changes to data layout and calling conventions discussed in Data Formats and Calling Conventions. Specifically, the compiler understands that symbols use the alternative linkage name, 64-bit types and bitfields have the opposite endianness, addresses belong to the opposite address alias space, and that the `MRF` and `MRB` registers may not be call-preserved. The compiler is then able to perform type checking and type conversions as necessary when the external functions and data are used.

If no pragma is used, the compiler assumes the declaration uses the char-size being used for the current compilation. If a function or variable is declared with the wrong char-size, the usual result is that the link step will fail due to unresolved symbols.

Type Coercions

The compiler will identify places where type coercions are necessary to convert between types for different char size regions. Where it is possible to do so, the compiler will automatically perform the necessary type coercions, while in cases where it is not possible to do so, a compile-time error or warning will be reported. For example, if you compile the following code with `-char-size-8`

```
#pragma word_addressed
extern long long ll; // big-endian long long
long long double_ll(void) {
   return 2 * ll;  // compiler swaps endianness to convert to
                   // little-endian long long and multiplies by 2
}
```

the compiler will swap the endianness of the word-addressed `long long`, which is stored in memory in big-endian format, to a byte-addressed `long long`, which the function multiplies by 2 and returns in registers in little-endian format. Therefore you do not need to perform such type coercions explicitly. Likewise, when built `-char-size-8`,

```
#pragma word_addressed
extern void func(int *); // expects word-addressed pointer
void alloc_and_call_func(void) {
   int * buffer = malloc(10 * sizeof(int));
   func(buffer);  // compiler converts parameter to
                 // word-addressed alias space prior to call
}
```

the compiler will emit code to convert the pointer `buffer` into the word-addressed alias space as expected by the callee `func`.

Values held in memory are not converted by the compiler. For example, the following code, when compiled with `-char-size-8`, results in a compile-time warning, as the type in memory pointed at by `buffer` (one or more little-endian `long longs`) does not match the type in memory expected by the callee (one or more big-endian `long longs`).

```
#pragma word_addressed
extern void my_func(long long *); // expects word-addressed
                                  // pointer to big-endian long long
void call_func(long long *buffer) {
   my_func(buffer);  // cc0167: {D} warning: argument of type
                     // "long long *" is incompatible
                     // with parameter of type
                     // "word_addressed long long *word_addressed"
}
```

This and other similar warnings signify that the code probably do not function correctly.

Simple Interfaces

The simplest way to construct an interface between code compiled with `-char-size-8` and code compiled with `-char-size-32` is to define a small set of functions and variables which should be accessible to code built with the opposite char-size, and prototype these in a header file enclosed in `#pragma word_addressed(push)` and `#pragma word_addressed(pop)`, or `#pragma byte_addressed(push)` and `#pragma byte_addressed(pop)`. To allow the compiler to insert all the necessary type conversions, within an interface region try to:

- Avoid use of pointers to types which differ between `-char-size-8` and `-char-size-32`. Specifically, avoid pointers to `char`s, `short`s, `long longs` and `long doubles`, and also avoid pointers to other pointers.
- Avoid pointers to bitfields.
- Avoid use of scalar types whose size is less than 32 bits. Specifically, do not use `char`, `short` or `bool` types within a `#pragma byte_addressed` interface region.
- Avoid use of pointers to functions.

Interfaces adhering to the above constraints can be constructed and used simply by enclosing the extern declarations in the `word_addressed` or `byte_addressed` pragmas. Constructing interfaces where the above constraints do not hold may require more advanced techniques. For more information, see Constructing Complex Byte-Addressed and Word-Addressed Interfaces.

Constructing Complex Byte-Addressed and Word-Addressed Interfaces

Mixed Char-Size Applications discussed the construction of simple interfaces between code compiled with `-char-size-8` and code compiled with `-char-size-32`. This section deals with construction of more advanced interfaces, which may contain pointers to types whose representation differs between `-char-size-8` and `-char-size-32` compilations.

To construct these interfaces, it is useful to understand

- Use of the keywords to explicitly specify char-size, discussed in The byte_addressed and word_addressed Keywords.
- Constructs that are prohibited in interface regions, discussed in Prohibited Constructs in Interface Regions.
- The constraints on using sub-word types in word-addressed code, discussed in Using Sub-Word Types in Word-Addressed Code.
- The behavior of typedefs in interface regions, discussed in Use of typedefs.
- How to use function pointers, discussed in Function Pointers.

The byte_addressed and word_addressed Keywords

The `byte_addressed` and `word_addressed` keywords (which may be optional prefixed by two underscores: `__byte_addressed` and `__word_addressed`) may be used to explicitly specify char-size, memory

space and endianness outside of interface regions. The `byte_addressed` keyword specifies that the type has the representation used in a compilation using `-char-size-8`. The `word_addressed` keyword specifies that the type has the representation used in a compilation using `-char-size-32`. The qualifier may be used on any scalar type, but only has an effect on types whose representation differs between `-char-size-8` and `-char-size-32` compilations. The qualified types are shown in the ***Types Explicitly Qualified With the byte_addressed and word_addressed Qualifiers*** table.

Table 2-26: Types Explicitly Qualified With the byte_addressed and word_addressed Qualifiers

Type	*Size in Bits - Compiled Either* **-char-size-8** or **-char-size-32**	*Notes*
byte_addressed char	8	
word_addressed char	32	
byte_addressed short	16	
word_addressed short	32	
byte_addressed int	32	byte_addressed has no effect on type
word_addressed int	32	word_addressed has no effect on type
byte_addressed long long	64	little-endian
word_addressed long long	64	big-endian
byte_addressed float	32	byte_addressed has no effect on type
word_addressed float	32	word_addressed has no effect on type
byte_addressed double	32 or 64	little-endian if 64-bit
word_addressed double	32 or 64	big-endian if 64-bit
byte_addressed long double	64	little-endian
word_addressed long double	64	big-endian
byte_addressed short fract	32	byte_addressed has no effect on type
word_addressed short fract	32	word_addressed has no effect on type
byte_addressed fract	32	byte_addressed has no effect on type
word_addressed fract	32	word_addressed has no effect on type
byte_addressed long fract	32	byte_addressed has no effect on type
word_addressed long fract	32	word_addressed has no effect on type
byte_addressed pointer (e.g. int * byte__addressed)	32	address in byte-addressed memory space
word_addressed pointer (e.g. int *word_addressed	32	address in word-addressed memory space

NOTE: The signed and unsigned variants of the integer types are the same sizes as the equivalent plain types. Note than byte_addressed and word_addressed have no effect on the int, float, and fract types. The size of the double type depends on use of the switches -double-size-32 and -double-size-64.

Within interface regions, explicit qualification overrides the implicit qualification resulting from the interface region. So, for example, if a function my_func is declared with:

```
#pragma word_addressed
int * byte_addressed my_func(void);
```

It is expected to return an address in the byte-addressed memory space.

The byte_addressed and word_addressed qualifiers allow you to use types pertaining to the opposite char-size within your code. This can help you to perform any explicit conversions necessary in order to use functions and data within an interface region. For example, consider a simple function, compiled with -char-size-32, that returns a long long result by storing it in a memory location passed from the caller:

```
void my_func(long long *p) {
   *p = 0x12345678abcdefLL;
}
```

This could be used from byte-addressed code by using the word_addressed qualifier:

```
#pragma word_addressed
void my_func(long long *p);

void my_func_wrapper(long long *p) {
  word_addressed long long ll;
  my_func(&ll);
  *p = ll;
}
```

Prohibited Constructs in Interface Regions

Interface regions are used to declare functions and data that are defined outside of the current source file, in translation units built with the opposite char-size. Consequently, there are a number of constraints on the constructs that can appear in an interface region:

- The compiler will produce an error if function definitions (i.e. function declarations which include their source implementation) or data definitions (i.e. data declarations that result in storage being allocated within the current translation unit) are found in an interface region.
- Word-addressed code cannot define or access values of less than 32 bits in size. A word_addressed interface therefore cannot contain variables of type less than 32 bits in size, or of struct, union or class types containing fields of less than 32 bits in size. For example, the compiler issues an error for the following declaration, as it declares a word-addressed variable less than 32 bits in size:

  ```
  #pragma word_addressed
  extern byte_addressed char c;
  ```

  However the following is a legal declaration, as it declares a pointer which is 32 bits in size:

```
#pragma word_addressed
extern byte_addressed char *p;
```

Likewise, functions in word-addressed regions cannot generally take parameters or return values of these types. A special exception is made to allow returning a value of type `byte_addressedchar`, `bool` or `short`, which is returned sign- or zero-extended in the return register `R0`.

There are no constraints on types that may be used in `byte_addressed` interface regions.

Using Sub-Word Types in Word-Addressed Code

Types of less than 32 bits in size, or `struct`, `union` or `class` types containing fields of less than 32 bits in size, cannot be used directly in code compiled with `-char-size-32`. The compiler will produce an error when compiling the following two code examples with `-char-size-32`:

```
#pragma byte_addressed
extern char c;
char fn(void) {
  return c; // error: cannot access 8-bit values in word_addressed code
}
```

and

```
byte_addressed short fn2(int x) {
  return (byte_addressed short)x; // error: cannot create 16-bit values
                                  // in word_addressed code
}
```

It is also illegal to perform pointer arithmetic on word-addressed pointers to types of size less than 32 bits, and `structs`, `unions` and `classes` containing these types, as a word-addressed pointer may not be able to represent the address. So the following is reported as an error by the compiler, when building with `-char-size-32`:

```
byte_addressed char *func(byte_addressed char *p) {
  return p+5; // error: can't increment word_addressed pointer by 5 bytes
}
```

while the following code produces an error when compiling with `-char-size-8`:

```
char *word_addressed func(char *word_addressed p) {
  return p+5; // error: can't increment word_addressed pointer by 5 bytes
}
```

Finally, when compiling with `-char-size-32`, the compiler also reports an error when performing pointer arithmetic on byte-addressed pointers to types of size less than 32 bits, and `structs`, `unions` and `classes` containing these types, as the compiler does not represent the size of these types, which may not be integral numbers of words:

```
byte_addressed char *byte_addressed func(byte_addressed char *byte_addressed p) {
  return p+5; // error: incrementing by 5 bytes when building with -char-size-32
}
```

These restrictions do not affect what declarations may appear in an interface region itself, but only what may be used within code using the interface definition.

Use of typedefs

`typedefs` can be used to declare simpler names for types of the opposite char-size. For example, you could declare

```
typedef byte_addressed char char8_t;
typedef word_addressed char char32_t;
```

When `typedefs` are used within interface regions, they retain the char-size of the interface region. So an equivalent way to define these types is to write

```
#pragma byte_addressed(push)
typedef char char8_t;
#pragma byte_addressed(pop)
#pragma word_addressed(push)
typedef char char32_t;
#pragma word_addressed(pop)
```

However a `typedef` with no explicit char-size (i.e. not defined within an interface region, and without a `byte_addressed` or `word_addressed` qualifier), when used within interface regions will take the prevailing char-size in the interface region:

```
typedef char char_t;
#pragma byte_addressed
extern char_t c1; // 8-bit char
#pragma word_addressed
extern char_t c2; // 32-bit char
```

Function Pointers

As discussed in Data Formats and Calling Conventions, functions built with `-char-size-8` and `-char-size-32` obey distinct calling conventions. Therefore calls through function pointers to functions built for the opposite char-size must use a function pointer type that specifies this. The easiest way to do this is to declare the function pointer type within the interface. For example

```
#pragma word_addressed(push)
extern void fn(int *);
typedef void (*wa_fn_ptr_t)(int);
#pragma word_addressed(pop)
wa_fn_ptr_t my_fn_pointer = fn; // ok - function pointer type
                                // and function used as initializer
                                // are both word-addressed
```

Alternatively, the `word_addressed` and `byte_addressed` keywords can be used to explicitly state the kind of function pointed at. In the following example, the use of `word_addressed` in the function pointer declaration declares the function pointed at as being in word-addressed code:

```
#pragma word_addressed
extern void fn(int *p);
void (word_addressed *my_fn_pointer)(int) = fn;
```

Using Byte-Addressed and Word-Addressed Interfaces with C++

Interface regions can be used in both C and C++ code. A number of C++ features, such as overloading, function templatization, exceptions, and run-time type identification, depend on the types of variables and function parameters. The way the `word_addressed` and `byte_addressed` type qualifiers are treated has an effect on these features.

Functions can be overloaded using the `byte_addressed` and `word_addressed` qualifiers, so long as the qualified type differs from the unqualified one. For example, you can create two versions of a function, one with a `byte_addressed long long` argument and another with a `word_addressed long long argument`, but you cannot do the same overloading using `byte_addressed int` and `word_addressed int`, as these two latter types are equivalent. Which types are affected by the `byte_addressed` and `word_addressed` qualifiers is discussed in Data Formats and Calling Conventions. Where an exact overloaded function match cannot be found, the compiler will call the closest match which may involve a conversion of parameters from `byte_addressed` to `word_addressed` types or vice versa.

Similarly, for template functions, use of template parameters that differ by the `word_addressed` or `byte_addressed` qualifiers will instantiate distinct copies of the template function, so long as the `word_addressed` and `byte_addressed` types differ.

A catch block will catch objects based on the bare type. For example, `catch (long long a)` will catch a thrown object of either `word_addressed long long` or `byte_addressed long long`. Therefore it is necessary to be careful to only throw objects of the address space expected by the `catch` block.

Run-time type identification also ignores the address-space qualifier. So `byte_addressed long long` and `word_addressed long long` will be identified as the same type.

Assembly Code

Assembly code can be called from C/C++ code if it is written to obey the calling conventions used in compiled code. Differences in the calling conventions between code compiled with `-char-size-8` and `-char-size-32` are discussed in Data Formats and Calling Conventions.

Use of assembly code in applications with byte addressing obeys many of the same constraints as C/C++ code. These are discussed in Mixed Char-Size Applications.

This section additionally covers ways in which assembly code written for word-addressed applications may be used in byte-addressed applications with minimum overhead.

- The differences between the operation of the byte-addressed address alias and the SW, NW and LW address alias spaces are discussed in Memory Alias Spaces Compared.
- Guidelines for how to use circular buffers in assembly code are given in Using Circular Buffers in Assembly Code.
- Declaring assembly functions for use in byte-addressed applications is discussed in Using Word-Addressed Assembly Code Within a Byte-Addressed Application.

- Using `#pragma no_stack_translation` to avoid converting the frame and stack pointers at the call-site is discussed in #pragma no_stack_translation.
- Writing char-size agnostic assembly code, which can be called directly from both byte-addressed and word-addressed parts of your application, is discussed in Char-Size Agnostic Assembly Code.
- Using byte-addressed data from assembly code is discussed in Defining Byte-Addressed Data in Assembly.
- Using `.IMPORT` in assembly files written for byte addressing is discussed in Byte-Addressed C structs in Assembly.

Memory Alias Spaces Compared

The byte-addressed alias space behaves differently from the SW and LW address aliases.

The latter may be used with standard load and store instructions, and the alias space of the address determines the size of the data accessed in memory. By contrast, use of the byte-addressed alias does not change the size of data accessed. Instead, access size is specified by the instruction used. Therefore 32 bits of data can be loaded from memory using either a word address or its equivalent byte address.

In addition, individual 8-bit bytes, or 16-bit shorts, can be accessed from a byte address by using a sub-word load or store instruction. The ***Behavior of Load and Modify Instructions for Different Memory Spaces of Index Register*** table shows this distinction, where each instruction accesses the same underlying physical memory location. Please note that the scaling behavior described applies to both offsets held in registers, and literal (constant) offsets.

Table 2-27: Behavior of Load and Modify Instructions for Different Memory Spaces of Index Register

Instruction	`i5` *in NW Space*	`i5` *in LW Space*	`i5` *in SW Space*	*i5 in BW Space*
`r0 = dm(m4, i5) (lw)`	Load 64 bits from address into registers `r0` and `r1`. Offset `m4` in 32-bit words.	Load 64 bits from address into registers `r0` and `r1`. Offset `m4` in 64-bit long words.	Load 16 bits from address into register `r0`, sign-extended to 32 bits if the `SSE` bit in `MODE1` is set. Offset `m4` in 16-bit short words.	Load 64 bits from address into registers `r0` and `r1`. Offset `m4` in 32-bit words.
`r0 = dm(m4, i5)`	Load 32 bits from address into register `r0`. Offset `m4` in 32-bit words.	Load 64 bits from address into registers `r0` and `r1`. Offset `m4` in 64-bit long words.	Load 16 bits from address into register `r0`, sign-extended to 32 bits if the `SSE` bit in `MODE1` is set. Offset `m4` in 16-bit short words.	Load 32 bits from address into register `r0`. Offset `m4` in 32-bit words.
`r0 = dm(m4, i5) (sw)`	Illegal	Illegal	Illegal	Load 16 bits from address into register `r0`, zero-extended to 32 bits. Offset `m4` in 16-bit short words.
`r0 = dm(m4, i5) (swse)`	Illegal	Illegal	Illegal	Load 16 bits from address into register `r0`, sign-extended to 32 bits. Offset `m4` in 16-bit short words.

Table 2-27: Behavior of Load and Modify Instructions for Different Memory Spaces of Index Register (Continued)

Instruction	`i5` *in NW Space*	`i5` *in LW Space*	`i5` *in SW Space*	*i5 in BW Space*
`r0 = dm(m4, i5) (bw)`	Illegal	Illegal	Illegal	Load 8 bits from address into register `r0`, zero-extended to 32 bits. Offset `m4` in 8-bit bytes.
`r0 = dm(m4, i5) (bwse)`	Illegal	Illegal	Illegal	Load 8 bits from address into register `r0`, sign-extended to 32 bits. Offset `m4` in 8-bit bytes.
`modify(i5, m4)`	Offset m4 unscaled (32-bit words).	Offset m4 unscaled (64-bit long words).	Offset m4 unscaled (16-bit short words).	Offset m4 unscaled (8-bit bytes).
`modify(i5, m4) (sw)`	Illegal	Illegal	Illegal	Offset m4 scaled by 2 (16-bit short words).
`modify(i5, m4) (nw)`	Offset m4 unscaled (32-bit words).	Illegal	Illegal	Offset m4 scaled by 4 (32-bit words).

Using Circular Buffers in Assembly Code

When using hardware support for circular buffering, through either `modify` instructions or the post-increment addressing mode, the value in the length register (`l0` through `l15`) is interpreted in terms of the modifier value scaling used by the memory access or `modify` instruction. In general, this is the same as the memory access size.

This has a number of implications when using circular buffers in assembly code.

- All memory accesses and modify instructions should use the same modifier scaling, and the length register should be specified in terms of that size. Specifically:
  - Accessing 8-bit quantities: Memory accesses `(bw)` or `(bwse)` qualified, `modify` instructions unqualified, length register counts number of 8-bit quantities in buffer.
  - Accessing 16-bit quantities: Memory accesses `(sw)` or `(swse)` qualified, `modify` instructions `(sw)` qualified, length register counts number of 16-bit quantities in buffer.
  - Accessing 32-bit quantities: Memory accesses unqualified, `modify` instructions `(nw)` qualified, length register counts number of 32-bit quantities in buffer.
  - Accessing 64-bit quantities: Memory accesses unqualified or `(lw)` qualified, `modify` instructions `(nw)` qualified, length register counts number of 32-bit quantities in buffer (i.e. twice the number of 64-bit quantities).

  For example, if you wish to access a circular buffer of one hundred 16-bit values in memory, then your length register should be set to the number of 16-bit elements in your circular buffer (100), and all memory accesses and `modify` instructions should use either the `(sw)` or `(swse)` qualifier.

- When modifying the frame or stack registers `i6` and `i7`, the `modify` instruction should always use the `(nw)` qualifier. This is because the length registers, `l6` and `l7`, are specified as the maximum size of the stack in number of 32-bit words.

Using Word-Addressed Assembly Code Within a Byte-Addressed Application

The rules for using word-addressed assembly code within a byte-addressed application are the same as for use of word-addressed C/C++ code. The recommended way is to leave your assembly code unchanged, and declare the function within a word-addressed interface region. For example

```
#pragma word_addressed
int my_word_addressed_assembly_function(int a, int b);
```

Correct prototyping will ensure the compiler performs the necessary type-checking and conversions when calling the function from C/C++ code.

#pragma no_stack_translation

One overhead of calling word-addressed code from byte-addressed code (or vice versa) is the need to transform the frame and stack pointer into the opposite address space at the call site. Typical code produced by the compiler will perform

```
  i7=b2w(i7);
  b7=b2w(b7);
  b6=b2w(b6);
  cjump _f (db); // call to word-addressed function
    dm(i7,m7)=r2;
    dm(i7,m7)=.LCJ0-1;
.LCJ0:
  b6=w2b(b6);
  b7=w2b(b7);
  i7=w2b(i7);
```

If your callee function is written in assembly code, you may be able to prove that it does not depend on the frame and stack pointers being in a particular address space. For this to be the case, all memory access to the stack must be to words or long words (using the `LW` qualifier), and all arithmetic on the frame and stack pointers must be achieved with the `(NW)` variant of the `modify` instruction. If this is true, you can apply the `no_stack_translation` pragma to your function declaration. The pragma tells the compiler to suppress the translation of the frame and stack pointers to the callee's address space before the call, and the translation back to the native address space after the call.

Char-Size Agnostic Assembly Code

A step further is to write assembly code that can be called directly from either word-addressed or byte-addressed code without the need to declare it in an interface region. The requirements for such a function are:

- all accesses to memory must access words or long words (using the `LW` qualifier)
- all pointer arithmetic must be performed using the `(NW)` variant of the `modify` instruction
- the code uses no bitfields or 64-bit floating-point or integer types

- the MRF and MRB registers are call-preserved

If your assembly code obeys these constraints, it can be called from either word-addressed or byte-addressed code without any frame pointer, stack pointer or operand conversions at the call site. In this case, you can add an additional entry point to the assembly function using the byte-addressed symbol name, and declare your function outside of any interface region.

```
_func: // word-addressed entry point
func.: // byte-addressed entry point
// function implementation here
._func.end:
.func..end:
```

Defining Byte-Addressed Data in Assembly

In assembly, byte-addressed data sections are introduced by supplying the BW type specifier to the .SECTION directive. Alternatively, the type specifier can just be omitted.

In byte-addressed sections, the directives shown in the ***Data Declaration Directives for Byte-Addressed Data*** table are supported for defining and initializing data items. The syntax of these directives is the same as for the .VAR directive.

Table 2-28: Data Declaration Directives for Byte-Addressed Data

.BYTE	Single-byte items
.BYTE2	Two-byte (short word) items
.BYTE4	Four-byte (normal word) items
.BYTE8	Eight-byte (long word) item

The .VAR directive itself behaves the same as the .BYTE4 directive when used in a byte-addressed data section.

Floating-point initializer values are encoded in single precision format when appearing in a .BYTE4 directive, and in double precision format when appearing in a .BYTE8 directive. Floating-point initializer values are not allowed in .BYTE and .BYTE2 directives.

ASCII strings in initializer values are represented as one item per character, meaning one byte per character in .BYTE directives, two bytes per character in .BYTE2 directives, and so on.

Initializer values in .BYTE2, .BYTE4 and .BYTE8 directives are written in little-endian byte order. Instructions cannot be placed into byte-addressed sections.

An example of use of the data declaration directives is below:

```
.section/bw data1;
.byte byte = 0x12;
.byte2 short = 0x1234;
.byte4 word = 0x12345678;
.byte4 single = 1.0;
```

```
.byte8 double = 2.0;
.byte string[] = 'Hello', 0;
```

Byte-Addressed C structs in Assembly

The assembler has the ability to import struct type definitions from C headers and instantiate and access such structs from assembly code. To this end, the assembler supports the same `-char-size-8` and `-char-size-32` switches as the compiler, whereby the default is `-char-size-8` for SHARC+ processors. These determine how the types in C headers are interpreted. They also determine where `.STRUCT` directives can be used to instantiate struct types: with `-char-size-8`, they can only be used in byte-addressed (`BW`) sections, whereas with `-char-size-32`, they can only be used in word-addressed (`DM`) sections.

The char-size switches determine how struct sizes and field offsets are calculated by the `SIZEOF()` and `OFFSETOF()` built-in functions, as well as in field accesses. This means that these should only be used in connection with pointers to the appropriate memory space, i.e. the byte-addressed space when using `-char-size-8` and the word-addressed space when using `-char-size-32`.

When using the `SIZEOF()` and `OFFSETOF()` builtin-in functions in modify, load and store operations in the byte-addressed space, it is important that the address scaling by the processor is taken into account. For loads and stores, scaling is by the size of the access, while for modifies it is dependent on the instruction, specified in the `(sw)` or `(nw)` flag. To ensure that the `SIZEOF()` and `OFFSETOF()` functions operate as expected in modify, load and store operations, the results should be adjusted as follows:

- For long-word and normal-word accesses, the result should be divided by 4. For example:

  ```
  r0 = dm(OFFSETOF(myStruct_t, myStructMember)/4, i0);
  ```

- For short-word accesses, the result should be divided by 2. For example:

  ```
  r0 = dm(OFFSETOF(myStruct_t, myStructMember)/2, i0) (sw);
  ```

No such adjustments are required when performing byte accesses or when accessing the word-addressed space.

See "Scaled Address Arithmetic" section of the SHARC+ Core Programming Reference for more information on address scaling.

Using Native Fixed-Point Types

This section provides an overview of the compiler's support for the native fixed-point type `fract`, defined in Chapter 4 of the *Extensions to support embedded processors* ISO/IEC draft technical report, Technical Report 18037.

Fixed-Point Type Support

A fixed-point data type is one where the radix point is at a fixed position. This includes the integer types (the radix point is immediately to the right of the least-significant bit). However, this section uses the term to apply exclusively to those that have a non-zero number of fractional bits - that is, bits to the right of the radix point.

The SHARC processor has hardware support for arithmetic on 32-bit fixed-point data types. For example, it is able to perform addition, subtraction and multiplication on 32-bit fractional values. However, the C language does not make it easy to express the semantics of the arithmetic that maps to the underlying hardware support.

To make it easier to use this hardware capability, and to facilitate expression of DSP algorithms that manipulate fixed-point data, the compiler supports a number of native fixed-point types whose arithmetic obeys the fixed-point semantics. This makes it easy to write high-performance algorithms that manipulate fixed-point data, without having to resort to compiler built-ins, or inline assembly.

An emerging standard for such fixed-point types is set out in Chapter 4 of the *Extensions to support embedded processors* ISO/IEC Technical Report 18037. CCES provides much of the functionality specified in that chapter, and the chapter is a useful reference that explains the subtleties of the semantics of the library functions and arithmetic operators. The following sections give an overview of these data types, the semantics of arithmetic using these types, and guidelines for how to write high-performance code using these types.

Native Fixed-Point Types

The keyword `_Fract` is used to declare variables of fixed-point type. The `_Accum` keyword, defined in the ISO/IEC Technical Report to specify fixed-point data types with an integer as well as a fractional part is not currently supported by CCES for SHARC processors. The `_Fract` keyword may also be used in conjunction with the type specifiers `short` and `long`, and `signed` and `unsigned`. There are therefore 6 fixed-point types available, although many of these are aliases for types of the same size and format.

By including the header file `stdfix.h`, the more convenient alternative spelling `fract` may be used instead of `_Fract`. This header file also provides prototypes for many useful functions and it is highly recommended that you include it in source files that use fixed-point types. Therefore, the discussion that follows uses the spelling `fract`, as does the rest of the CCES documentation.

The formats of the fixed-point types are given in the ***Data Storage Formats, Ranges, and Sizes of the Native Fixed-Point Types*** table. In the ***Representation*** column of the table, the number after the point indicates the number of fractional bits, while the number before the point refers to the number of integer bits, including a sign bit when it is preceded by "s". Signed types are in two's complement form. The range of values that can be represented is also given in the table. Note that the bottom of the range can be represented exactly, whereas the top of the range cannot - only the value one bit less than this limit can be represented. The value returned by the `sizeof` operator depends on whether the `-char-size-8` or `-char-size-32` switch is being used.

Table 2-29: Data Storage Formats, Ranges, and Sizes of the Native Fixed-Point Types

Type	*Representation*	*Range*	*sizeof Returns*
`short fract`	s1.31	[-1.0,1.0)	1 (`-char-size-32`) or 4 (`-char-size-8`)
`fract`	s1.31	[-1.0,1.0)	1 (`-char-size-32`) or 4 (`-char-size-8`)
`long fract`	s1.31	[-1.0,1.0)	1 (`-char-size-32`) or 4 (`-char-size-8`)
`unsigned short fract`	0.32	[0.0,1.0)	1 (`-char-size-32`) or 4 (`-char-size-8`)

Table 2-29: Data Storage Formats, Ranges, and Sizes of the Native Fixed-Point Types (Continued)

Type	*Representation*	*Range*	*sizeof Returns*
`unsigned fract`	0.32	[0.0,1.0)	1 (`-char-size-32`) or 4 (`-char-size-8`)
`unsigned long fract`	0.32	[0.0,1.0)	1 (`-char-size-32`) or 4 (`-char-size-8`)

The Technical Report also defines a `_Sat` (alternative spelling `sat`) type qualifier for the fixed-point types. This stipulates that all arithmetic on fixed-point types shall be saturating arithmetic (that is, that the result of arithmetic that overflows the maximum or minimum value that can be represented by the type shall saturate at the largest or smallest representable value). When the `sat` qualifier is not used, the standard says that arithmetic that overflows may behave in an undefined manner. CCES accepts the `sat` qualifier for compatibility but will always produce code that saturates on overflow whether the `sat` qualifier is used or not. This gives maximum reproducibility of results and permits code to be written without worrying about obtaining unexpected results on overflow.

Native Fixed-Point Constants

Fixed-point constants may be specified in the same format as for floating-point constants, inclusive of any decimal or binary exponent. For more information on these formats, refer to strtofxfx. Suffixes are used to identify the type of constants. The `stdfix.h` header also declares macros for the maximum and minimum values of the fixed-point types. See the ***Fixed-Point Type Constant Suffixes and Macros*** table for details of the suffixes and maximum and minimum fixed-point values.

Table 2-30: Fixed-Point Type Constant Suffixes and Macros

Type	*Suffix*	*Example*	*Minimum Value*	*Maximum Value*
`short fract`	hr	0.5hr	SFRACT_MIN	SFRACT_MAX
`fract`	r	0.5r	FRACT_MIN	FRACT_MAX
`long fract`	lr	0.5lr	LFRACT_MIN	LFRACT_MAX
`unsigned short fract`	uhr	0.5uhr	0.0uhr	USFRACT_MAX
`unsigned fract`	ur	0.5ur	0.0ur	UFRACT_MAX
`unsigned long fract`	ulr	0.5ulr	0.0ulr	ULFRACT_MAX

A Motivating Example

Consider a very simple example-pairwise addition of two sets of fractional values, saturating at the largest or smallest fractional value if the addition overflows. Assume that the data consist of vectors of 32-bit values, representing values in the range [-1.0,1.0). Then it is natural to write:

Example

```
#include <stdfix.h>

void pairwise_add(fract *out, const fract *a, const fract *b, int n)
{
```

```
   int i;
   for (i = 0; i < n; i++)
      out[i] = a[i] + b[i];
}
```

The above code shows that it is easy to express algorithms that manipulate fixed-point data and perform saturation on overflow without needing to find special ways to express these semantics through integer arithmetic.

Fixed-Point Arithmetic Semantics

The semantics of fixed-point arithmetic according to the Technical Report are as follows:

1. If a binary operator has one floating-point operand, the other operand is converted to floating-point and the operator is applied to two floating-point operands to give a floating-point result.
2. If the operator has two fixed-point operands of different signedness, convert the unsigned one to signed without changing its size. (However, see also FX_CONTRACT.)
3. Deduce the result type. The result type is the operand type of highest rank. Rank increases in the following order: `short fract`, `fract`, `long fract` (or their unsigned equivalents). An operator with only one fixed-point operand produces a result of this fixed-point type. (An exception is the result of a comparison, which gives a boolean result.)
4. The result is the mathematical result of applying the operator to the operand values, converted to the result type deduced in step 3. In other words, the result is as if it was computed to infinite precision before converting this result to the final result type.

The conversions between different types are discussed in Data Type and Data Type Sizes.

Data Type Conversions and Fixed-Point Types

The rules for conversion to and from fixed-point types are as follows:

1. When converting to a fixed-point type, if the value of the operand can be represented by the fixed-point type, the result is this value. If the operand value is out of range of the fixed-point type, the result is the closest fixed-point value to the operand value. In other words, conversion to fixed-point saturates the operand's mathematical value to the fixed-point type's range. If the operand value is within the range of the fixed-point type, but cannot be represented exactly, the result is the closest value either higher or lower than the operand value. For more information, see Rounding Behavior.
2. When converting to an integer type from a fixed-point type, the result is the integer part of the fixed-point type. The fractional part is discarded, so rounding is towards zero; both `(int)(0.9r)` and `(int)(-0.9r)` give `0`.
3. When converting to a floating-point type, the result is the closest floating-point value to the operand value.

These rules have some important consequences of which you should be aware:

CAUTION: Conversion of an integer to a fractional type is only useful when the integer is -1, 0, or 1. Any other integer value will be saturated to the fractional type. So a statement:

```
fract f = 0x40000000; // try to assign 0.5 to f
```

does not assign `0.5` to `f`, but results in `FRACT_MAX`, because `0x40000000` is an integer greater than 1. Instead, use:

```
fract f = 0.5r;
```

- or -

```
fract f = 0x40000000p-31r;
```

Note that the second format above uses the binary exponent syntax available for fixed-point constants; specifically the value `0x40000000` is scaled by 2^{-31}.

CAUTION: Assignment of a fractional value to an integer yields zero unless the fractional value is `-1.0`. Assignment of an unsigned fractional value to an integer always results in zero.

Compiler warnings will be produced to aid in the diagnosis of problems where these conversions are likely to produce unexpected results.

Bit-Pattern Conversion Functions: bitsfx and fxbits

The `stdfix.h` header file provides functions to convert a bit pattern to a fixed-point type and vice versa. These functions are particularly useful for converting between native types (`fract`, `unsigned fract`) and integer bit-patterns representing these values.

For each fixed-point type, a corresponding integer type is declared, which is big enough to hold the bit pattern for the fixed-point type. These are `int_fx_t`, where *fx* is one of hr, r, or lr, and `uint_fx_t` where *fx* is one of uhr, ur, or ulr.

To convert a fixed-point type to a bit pattern, use the `bitsfx` family of functions. *fx* may be any of `hr`, `r`, `lr`, `uhr`, `ur`, or `ulr`. For example, using the prototype

```
uint_ur_t bitsur(unsigned fract);
```

you can write

```
#include <stdfix.h>
unsigned fract f;
uint_ur_t f_bit_pattern;

void foo(void) {
   f = 0.5ur;
   f_bit_pattern = bitsur(f);   // gives 0x80000000
}
```

Similarly, to convert to a fixed-point type from a bit pattern, use the `fxbits` family of functions. So, to convert from a `int_lr_t` to a `long fract`, use:

```
#include <stdfix.h>
#include <fract.h>
int_lr_t f32;
long fract lf;

void foo(void) {
   f32 = 0x40000000;      // that's 0.5
   lf = lrbits(f32);      // gets 0.5lr as expected
}
```

For more information, see Fixed-Point Type Support.

Arithmetic Operators for Fixed-Point Types

You can use the +, -, *, and / operators on fixed-point types, which have the same meaning as their integer or floating-point equivalents, aside from any overflow or rounding semantics. As discussed in Fixed-Point Type Support, fixed-point operations that overflow give results saturated at the highest or lowest fixed-point value. Rounding is discussed in Rounding Behavior.

You can use << to shift a fixed-point value up by a positive integer shift amount less than the fixed-point type size in bits. This gives the same result as multiplication by a power of 2, including overflow semantics:

```
#include <stdfix.h>
fract f1, f2;

void foo1(void) {
   f1 = 0.125r;
   f2 = f1 << 2;      // gives 0.5r
}

void foo2(void) {
   f1 = -0.125r;
   f2 = f1 << 10;      // gives -1.0r
}
```

You can also use >> to shift a fixed-point value down by an integer shift amount in the same range. This is defined to give the same result as division by a power of 2, including any rounding behavior:

```
#include <stdfix.h>
fract f1, f2;

void foo1(void) {
   f1 = 0.5r;
   f2 = f1 >> 2;      // gives 0.125r
}

void foo2(void) {
   f1 = 0x00000003p-31r;
   f2 = f1 >> 2; // gives 0x00000000p-31r when rounding
                 // mode is truncation
```

```
                  // and 0x00000001p-31r when rounding
                  // mode is biased or unbiased
}
```

Any of these operators can be used in conjunction with assignment, for example:

```
#include <stdfix.h>
fract f1, f2;

void foo1(void) {
   f1 = 0.2r;
   f2 = 0.3r;
   f2 += f1;
}
```

In addition, there are a number of unary operators that may be used with fixed-point types. These are:

- `++` Equivalent to adding integer 1
- `--` Equivalent to subtracting integer 1
- `+` Unary plus, equivalent to adding value to 0.0 (no effect)
- `-` Unary negate, equivalent to subtracting value from 0.0
- `!` 1 if equal to 0.0; 0 otherwise

FX_CONTRACT

Consider the example of a multiplying a signed `fract` by an unsigned one:

```
fract f;
unsigned fract uf;
f = f * uf;
```

According to the rules of fixed-point arithmetic, then since the two `fract` operands differ in signedness, the unsigned one is first converted to `signed fract`, and subsequently the two s1.31 operands are multiplied together to yield an s1.31 result. So the rules say that it should be equivalent to writing:

```
fract tmp = uf;
f = f * tmp;
```

However, this means that one bit of precision is lost from the unsigned operand before the multiplication. SHARC processors, however, have hardware support for multiplying two fractional operands of opposite signedness together directly, which involve no loss in precision. Use of this support is both more precise and more efficient.

For convenience, the compiler can do this step for you, using a mode known as `FX_CONTRACT`. The name `FX_CONTRACT` is used as the behavior is similar to that of `FP_CONTRACT` in C99. When `FX_CONTRACT` is on, the compiler may keep intermediate results in greater precision than that specified by the Technical Report. In other words, it may choose not to round away extra bits of precision or to saturate an intermediate result unnecessarily. More precisely, the compiler keeps the intermediate result in greater precision when:

- Maintaining the higher-precision intermediate result will be more efficient-it maps better to the underlying hardware.
- The intermediate result is not stored back to any named variable.
- No explicit casts convert the type of the intermediate result.

In other words,

```
f = f * uf;
```

will result in an instruction that multiplies the signed and unsigned operands together directly, but

```
f = f * (fract)uf;
```

- or -

```
fract tmp = uf;
f = f * tmp;
```

will both force the unsigned operand to be converted to `fract` type before the multiplication.

By default, the compiler permits `FX_CONTRACT` behavior. The `FX_CONTRACT` mode can be controlled with a pragma (see also #pragma FX_CONTRACT {ON|OFF}) or with command-line switches, `-fx-contract` and `-no-fx-contract`. The pragma can be used at file scope or within functions. It obeys the same scope rules as #pragma FX_ROUNDING_MODE {TRUNCATION|BIASED|UNBIASED} discussed in the Setting the Rounding Mode example.

Rounding Behavior

Some fixed-point operations are also affected by rounding. For example, multiplication of two fractional values to produce a fractional result of the same size requires discarding a number of bits of the exact result. For example, `s1.31 * s1.31` produces an exact `s2.62` result. This is saturated to `s1.62` and the thirty-one least-significant bits must be discarded to produce an `s1.31` result.

By default, any bits that must be discarded are truncated-in other words, they are simply chopped off the end of the value. For example:

```
#include <stdfix.h>
fract f1, f2, prod;

void foo(void) {
   f1 = 0x3ffffffp-31r;
   f2 = 0x10000000p-31r;
   prod = f1 * f2;  // gives 0x007fffffp-31r, discarding
                    // least-significant bits 0xe0000000
}
```

This is equivalent to always rounding down toward negative infinity. It tends to produce results whose accuracy tends to deteriorate as any rounding errors are generally in the same direction and are compounded as the calculations proceed.

If this does not give you the accuracy you require, you can use either biased or unbiased round-to-nearest rounding. The compiler supports pragmas and switches to control the rounding mode. In the biased or unbiased rounding modes, the above product will be rounded to the nearest value that can be represented by the result type, so the final result will be `0x00800000p-31r`.

The difference between biased and unbiased rounding occurs when the value to be rounded lies exactly half-way between the two closest values that can be represented by the result type. In this case, biased rounding will always round toward the greater of the two values (applying saturation if this rounding overflows) whereas unbiased rounding will round toward the value whose least-significant bit is zero. For example:

```
#include <stdfix.h>
fract f1, f2, prod;

void foo1(void) {
   f1 = 0x00008000p-31r;
   f2 = 0x34568000p-31r;
   prod = f1 * f2; // gives 0x3456p-31r in unbiased rounding
                   // mode, but 0x3457p-31r in biased rounding
                   // mode
}

void foo2(void) {
   f1 = 0x00008000p-31r;
   f2 = 0x34578000p-31r;
   prod = f1 * f2; // gives 0x3458p-31r in both unbiased
                   // and biased rounding modes
}
```

In general, unbiased rounding is more costly than biased rounding in terms of cycles, but yields a more accurate result since rounding errors in the half-way case are not all in the same direction and therefore are not compounded so strongly in the final result.

The rounding discussed here only affects operations that yield a fixed-point result. Operations that yield an integer result round toward zero. There are also a few exceptions to the rounding rules:

- Conversion of a floating-point value to a fixed-point value rounds towards zero.
- The `roundfx`, `strtofxfx`, and `fxdivi` functions always perform unbiased rounding. They do not support the truncation rounding mode.

Details of how to set rounding mode are given in Setting the Rounding Mode.

Arithmetic Library Functions

The `stdfix.h` header file also declares a number of functions that permit useful arithmetic operations on combinations of fixed-point and integer types. These are the divifx, idivfx, fxdivi, mulifx, absfx, roundfx, countlsfx, and strtofxfx families of functions.

divifx

The `divifx` functions, where *fx* is one of `r`, `lr`, `ur`, or `ulr`, allow division of an integer value by a fixed-point value to produce an integer result. If you write

```
#include <stdfix.h>
fract f;
int i, quo;

void foo(void) {
   // BAD: division of int by fract gives fract result, not int
   f = 0.5r;
   i = 2;
   quo = i / f;
}
```

then the result of the division is a `fract` whose integer part is stored in the variable `quo`. This means that the value of `quo` is zero, as the division overflows and thus produces a fractional result that is nearly one.

To get the desired result, write

```
#include <stdfix.h>
fract f;
int i, quo;

void foo(void) {
   // GOOD: uses divifx to give integer result
   f = 0.5r;
   i = 2;
   quo = divir(i, f);
}
```

which stores the value 4 into the variable `quo`.

idivfx

The `idivfx` functions, where *fx* is one of `r`, `lr`, `ur`, or `ulr`, allow division of a fixed-point value by a fixed-point value to produce an integer result. If you write:

```
#include <stdfix.h>
fract f1, f2;
int quo;

void foo(void) {
  // BAD: division of two fracts gives fract result, not int
  f1 = 0.5r;
  f2 = 0.25r;
  quo = f1 / f2;
}
```

then the result of the division is a `fract` whose integer part is stored in the variable `quo`. This means that the value of `quo` is zero, as the division overflows and thus produces a fractional result that is nearly one.

To get the desired result, write:

```
#include <stdfix.h>
fract f1, f2;
int quo;

void foo(void) {
  // GOOD: uses idivfx to give integer result
  f1 = 0.5r;
  f2 = 0.25r;
  quo = idivr(f1, f2);
}
```

which stores the value 2 into the variable `quo`.

fxdivi

The `fxdivi` functions, where *fx* is one of `r`, `lr`, `ur`, or `ulr`, allow division of an integer value by an integer value to produce a fixed-point result. If you write:

```
#include <stdfix.h>
int i1, i2;
fract quo;

void foo(void) {
  // BAD: division of int by int gives int result, not fract
  i1 = 5;
  i2 = 10;
  quo = i1 / i2;
}
```

then the result of the division is an integer which is then converted to a `fract` to be stored in the variable `quo`. This means that the value of `quo` is zero, as the division is rounded to integer zero and then converted to `fract`.

To get the desired result, write:

```
#include <stdfix.h>
int i1, i2;
fract quo;

void foo(void) {
  // GOOD: uses fxdivi to give fract result
  i1 = 5;
  i2 = 10;
  quo = rdivi(i1, i2);
}
```

which stores the value 0.5 into the variable `quo`.

mulifx

The `mulifx` functions, where *fx* is one of `r`, `lr`, `ur`, or `ulr`, allow multiplication of an integer value by a fixed-point value to produce an integer result. If you write:

```
#include <stdfix.h>
int i, prod;
fract f;

void foo(void) {
  // BAD: multiplication of int by fract produces fract result, not int
  i = 50;
  f = 0.5r;
  prod = i * f;
}
```

then the result of the multiplication is a `fract` whose integer part is stored in the variable `prod`. This means that the value of `prod` is zero, as the multiplication overflows and thus produces a fractional result that is nearly one.

To get the desired result, write

```
#include <stdfix.h>
int i, prod;
fract f;

void foo(void) {
  // GOOD: uses mulifx to give integer result
  i = 50;
  f = 0.5r;
  prod = mulir(i, f);
}
```

which stores the value 25 into the variable `prod`.

absfx

The `absfx` functions, where *fx* is one of `hr`, `r`, or `lr`, compute the absolute value of a fixed-point value.

In addition, you can also use the type-generic macro `absfx()`, where the operand type can be any of the signed fixed-point types.

roundfx

The `roundfx` functions, where *fx* is one of `hr`, `r`, `lr`, `uhr`, `ur`, or `ulr`, take two arguments. The first is a fixed-point operand whose type corresponds to the name of the function called. The second gives a number of fractional bits. The first operand is rounded to the number of fractional bits given by the second operand. The second operand must specify a value between 0 and the number of fractional bits in the type. Rounding is unbiased to-nearest, and saturation occurs when the value exceeds that which can be represented in the requested number of fractional bits.

```
#include <stdfix.h>
fract f, rnd;

void foo1(void) {
  f = 0x45608100p-31r;
  rnd = roundr(f, 15); // produces 0x45610000p-31r;
}
```

```
void foo2(void) {
  f = 0x7fff9034p-31r;
  rnd = roundr(f, 15); // produces 0x7fffffffp-31r;
}
```

In addition, you can also use the type-generic macro `roundfx()`, where the first operand type can be any of the signed fixed-point types.

countlsfx

The `countlsfx` functions, where *fx* is one of `hr`, `r`, `lr`, `uhr`, `ur`, or `ulr`, return the largest integer value `k` such that its operand, when shifted up by `k`, does not overflow. For zero input, the result is the size in bits of the operand type.

```
#include <stdfix.h>
int scal1, scal2;

void foo(void) {
  scal1 = countlsr(-0.1r);     // gives 3, because
                               // -0.1r<<3 = -0.8r
  scal2 = countlsur(0.1ur);    // gives 3, because
                               // 0.1ur<<3 = 0.8ur
}
```

In addition, you can also use the type-generic macro `countlsfx()`, where the operand type can be any of the signed fixed-point types.

strtofxfx

The `strtofxfx` functions, where the second *fx* is one of `hr`, `r`, `lr`, `uhr`, `ur`, or `ulr`, parse a string representation of a fixed-point number and return a fixed-point result. They behave similarly to `strtod`, and accept input in the same format.

Fixed-Point I/O Conversion Specifiers

The `printf` and `scanf` families of functions support conversion specifiers for the fixed-point types. These are given in the ***I/O Conversion Specifiers for the Fixed-Point Types*** table. Note that the conversion specifier for the signed types, `%r`, is lowercase while the one for the unsigned types, `%R`, is uppercase.

When used with the `scanf` family of functions, these conversion specifiers accept input in the same format as consumed by the `strtofxfx` (for more information, see strtofxfx) functions, which is the same as that accepted for `%f`.

Table 2-31: I/O Conversion Specifiers for the Fixed-Point Types

Type	*Conversion Specifier*
short fract	%hr

Table 2-31: I/O Conversion Specifiers for the Fixed-Point Types (Continued)

Type	*Conversion Specifier*
`fract`	`%r`
`long fract`	`%lr`
`unsigned short fract`	`%hR`
`unsigned fract`	`%R`
`unsigned long fract`	`%lR`

When used with the `printf` family of functions, fixed-point values are printed:

- As hexadecimal values by default, or when using the Lite version of the CCES I/O library. For example,

  ```
  printf("fract: %r\n", 0.5r);  // prints   fract: 40000000
  ```

- Like floating-point values when linking with the version of the CCES I/O library with full fixed-point support, using the `-flags-link -MD__LIBIO_FX` switch. For example,

  ```
  printf("fract: %r\n", 0.5r);  // prints   fract: 0.500000
  ```

Optional precision specifiers are accepted that control the number of decimal places printed, and whether a trailing decimal point is printed. However, these will have no effect unless the version of the CCES I/O library with full fixed-point support is being used. For more information, see the run-time library manual for SHARC processors.

Setting the Rounding Mode

As discussed in Rounding Behavior, these rounding modes are supported for fixed-point arithmetic:

- Truncation (this is the default rounding mode)
- Biased round-to-nearest rounding
- Unbiased round-to-nearest rounding

To set the rounding mode, use a pragma or a compile-time switch.

The following compile-time switches control rounding behavior:

- `-fx-rounding-mode-biased`
- `-fx-rounding-mode-truncation`
- `-fx-rounding-mode-unbiased`

The given rounding mode will then be the default for the whole of the source file being compiled.

You can also use a pragma to allow finer-grained control of rounding. The pragmas are:

- `#pragma FX_ROUNDING_MODE TRUNCATION`
- `#pragma FX_ROUNDING_MODE BIASED`
- `#pragma FX_ROUNDING_MODE UNBIASED`

If one of these pragmas is applied at file scope, it applies until the end of the translation unit or until another pragma at file scope changes the rounding mode.

If one of these pragmas is applied within a compound statement (that is, within a block enclosed by braces), the pragma applies to the end of the compound statement where it is specified. The rounding mode will return to the outer scope rounding mode on exit from the compound statement. An example of how to use these pragmas is given in the following example.

Example. Use of #pragma FX_ROUNDING_MODE to Control Rounding of Arithmetic on Fixed-Point Types

```
#include <stdfix.h>
#pragma FX_ROUNDING_MODE BIASED

fract my_func(void) {
    // rounding mode here is biased
    {
        #pragma FX_ROUNDING_MODE UNBIASED
        // rounding mode here is unbiased
    }
    // rounding mode here is biased
}
#pragma FX_ROUNDING_MODE TRUNCATION
fract my_func2(void) {
    // rounding mode here is truncation
}
```

For more information, see #pragma FX_ROUNDING_MODE {TRUNCATION|BIASED|UNBIASED}.

Language Standards Compliance

The compiler supports code that adheres to the ISO/IEC 9899:1990 C standard, ISO/IEC 9899:1999 C standard, ISO/IEC 14882:2003 C++ standard and the ISO/IEC 14882:2011 C++ standard.

- C Mode
- C++ Mode

The compiler's level of conformance to the applicable ISO/IEC standards is validated using commercial test-suites from Plum Hall, Perennial and Dinkumware.

C Mode

The compiler shall compile any program that adheres to a freestanding implementation of the ISO/IEC 9899:1990 C standard, but it does not prohibit the use of language extensions (C/C++ Compiler Language Extensions) that are compatible with the correct translation of standard-conforming programs. To enable this mode the `-c89` switch should be used.

The compiler does support the C99 keyword `_Complex` but not `_Imaginary`. The ISO/IEC 9899:1990 C standard library provided in C89 mode is used in C99 mode. This is the default mode (`-c99`).

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

- -const-strings
- -double-size[-32|-64]
- -enum-is-int. See also Enumeration Constants That Are Not int Type.

The language extensions cannot be disabled to ensure strict compliance to the language standards. However, when compiling for MISRA-C Compiler compliance checking, language extensions are disabled.

When the -c89 switch is used, these extensions already include many of the ISO/IEC 9899:1999 standard features. The following features are only available in C99 mode.

- Type qualifiers may appear more than once in the same specifier-qualifier-list.
- __func__ predefined identifier is supported.
- Universal character names (\u and \U) are accepted.
- The use of function declarations with non-prototyped parameter lists are faulted.
- The first statement of a for-loop can be a declaration, not just restricted to an expression.
- Type qualifiers and static are allowed in parameter array declarators.

C++ Mode

The compiler shall compile any program that adheres to a freestanding implementation of the ISO/IEC 14882:2003 C++ standard, but it does not prohibit the use of language extensions (C/C++ Compiler Language Extensions) that are compatible with the correct translation of standard-conforming programs. The Library provided in C++ mode is a proper subset of the full Standard C++ Library and is designed specifically for the needs of the embedded market.

Using the -c++11switch, support for many of the ISO/IEC 14882:2011 C++ standard language features can be enabled. However, there is no support for library features that are new to ISO/IEC 14882:2011. The key language features supported by the -c++11 switch are (please refer to the ISO/IEC 14882:2011 standard for detail on usage):

- The static_assert construct
- friend class syntax is extended to all nonclass types
- Mixed string literal concatenations
- The C99-style _Pragma operator
- An explicit instantiation maybe prefixed with the extern keyword
- The keyword auto can be used as a type specifier
- The keyword decltype

- Scoped enumeration types (defined with the keyword sequence `enum class`)
- Lambdas
- Rvalue referemces
- Ref-qualifiers
- Functions can be "`deleted`"
- Special member functions can be explitly "`defaulted`"
- Move constructors and move assignment operators
- Conversion functions can be marked `explicit`
- The keyword `nullptr` can be used as both a null pointer constant and a null pointer-to-member constant
- The context-sensitive keyword `final`
- Alias and alias template declarations
- Variadic templates
- U-literals and the `char16_t` and `char32_t` keywords
- Inline namespaces
- Initializer lists
- The `noexcept` specifier and keyword
- Range-based "`for`" loops
- Non-static data member initializers
- `constexpr` keyword
- Unrestricted unions
- Delegating constructors
- Ref-qualifiers on member functions
- Raw strings
- UTF-8 strings

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

- `-no-friend-injection`
- `-no-anach`
- `-no-implicit-inclusion`
- `-std-templates`

In addition, the best possible standard conformance is achieved using the following non-default switches:

- `-eh`
- `-const-strings`
- `-double-size[-32|-64]`
- `-rtti`

MISRA-C Compiler

This section provides an overview of MISRA-C compiler and MISRA-C 2004 Guidelines.

MISRA-C Compiler Overview

The Motor Industry Software Reliability Association (MISRA) in 1998 published a set of guidelines for the C programming language to promote best practice in developing safety related electronic systems in road vehicles and other embedded systems. The latest release of MISRA-C:2004 has addressed many issues raised in the original guidelines specified in MISRA-C:1998. Complex rules are now split into component parts. There are 121 mandatory and 20 advisory rules. The compiler issues a discretionary error for mandatory rules and a warning for advisory rules. More information on MISRA-C can be obtained at www.misra.org.uk/.

The compiler detects violations of the MISRA-C rules at compile-time, link-time and runtime. It has full support for the MISRA-C 2004 Guidelines. The majority of MISRA-C rules are easy to interpret.

Those that require further explanation can be found in Rules Descriptions.

As a documented extension, the compiler supports the type qualifiers `__pm` and `__dm` (see Dual Memory Support Keywords (pm dm)) and the integral types `long long` and `unsigned long long`. In addition, for processors that support byte-addressing (shown in Processor Features, the ***Processors Supporting Byte-Addressing*** table), the type qualifiers `__word_addressed` and `__byte_addressed` are supported (see The byte_addressed and word_addressed Keywords).

No other language extensions are supported when MISRA checking is enabled. Common extensions, such as the keywords `section` and `inline`, are not allowed in the MISRA-C mode, but the same effects can be achieved by using pragmas #pragma section/#pragma default_section and #pragma inline. Rules can be suppressed by the use of command-line switches or the MISRA-C extensions to Diagnostic Control Pragmas.

NOTE: The run-time checking that is used for validating a number of rules should not be used in production code. The cost of detecting these violations is expensive in both run-time performance and code size. A subset of these run-time checks can also be enabled when MISRA-C is not enabled. For more information, see Run-Time Checking.

Refer to the ***C Mode (MISRA) Compiler Switches*** table in Compiler Command-Line Switches for more information.

MISRA-C Compliance

The MISRA-C:2004 Guidelines Forum (visit http://www.misra.org.uk) is an essential reference for ensuring that code developed or requiring modification complies to these guidelines. A rigorous checking tool such as this compiler makes achieving compliance a lot easier than using a less capable tool or simply relying on manual reviews of the code. The MISRA-C:2004 Guidelines Forum describes a compliance matrix that a developer uses to ensure that each rule has a method of detecting the rule violation. A compliance checking tool is a vital component in detecting rule violations. It is recognized in the guidelines document that in some circumstances it may be necessary to deviate from the given rules. A formal procedure has to be used to authorize these deviations rather than an individual programmer having to deviate at will.

Using the Compiler to Achieve Compliance

The CCES compiler is one of the most comprehensive MISRA-C: 2004 compliance checking tools available. The compiler has various command-line switches and Diagnostic Control Pragmas to enable you to achieve MISRA-C: 2004 compliance.

During development, it is recommended that the application is built with maximum compliance enabled.

Use the `-misra-strict` command-line switch to detect the maximum number of rule violations at compile-time. However, if existing code is being modified, using `-misra-strict` may result in a lot of errors and warnings. The majority are usually common rule violations that are mainly advisory and typically found in header files as a result of macro expansion. These can be suppressed using the `-misra` command-line switch. This has the potential benefit of focusing change on individual source file violations, before changing headers that may be shared by more than one project.

The `-misra-no-cross-module` command-line switch disables checking rule violations that occur across source modules. During development, some external variables may not be fully utilized and rather than add in artificial uses to avoid rule violations, use this switch.

The `-misra-no-runtime` command-line switch disables the additional run-time overheads imposed by some rules. During development these checks are essential in ensuring code executes as expected. Use this switch in release mode to disable the run-time overheads.

You can use the `-misra-testing` command-line switch during development to record the behavior of executable code. Although the MISRA-C: 2004 Guidelines do not allow library functions, such as those that are defined in header `<stdio.h>`, it is recognized that they are an essential part of validating the development process.

During development, it is likely that you will encounter areas where some rule violations are unavoidable. In such circumstances you should follow the procedure regarding rule deviations described in the MISRA-C: 2004 Guidelines Forum. Use the `-Wmis_suppress` and `-Wmis_warn` *`switches`* to control the detection of rule violations for whole source files. Finer control is provided by the diagnostic control pragmas. These pragmas allow you to suppress the detection of specified rule violations for any number of C statements and declarations.

Example

```
#include <misra_types.h>
#include <def21266.h>
#include "proto.h" /* prototype for func_state and my_state */
```

```
int32_t func_state(int32_t state)
{
   return state & TIMOD1;
   /* both operands signed, violates rule 12.7 */
}

#define my_flag 1

int32_t my_state(int32_t state)
{
   return state & my_flag;
   /* both operands signed, violates rule 12.7 */
}
```

In the above example, `<def21266.h>` uses signed masks and signed literal values for register values. The code is meaningful and trusted in this context. You may suppress this rule and document the deviation in the code. For code violating the rule that is not from the system header, you may wish to rewrite the code:

```
#include <misra_types.h>
#include <def21266.h>
#include "proto.h" /* prototype for func_state and my_state */

#ifdef _MISRA_RULES
#pragma diag(push)
#pragma diag(suppress:misra_rule_12_7:"Using the def file is
a safe and justified deviation for rule 12.7")

#endif  /* _MISRA_RULES */

int32_t func_state(int32_t state)
{
   return state & TIMOD11;  /* both operands signed, violates rule 12.7 */
}

#ifdef _MISRA_RULES
   #pragma diag(pop)  /* allow violations of 12.7 to be detected again */
#endif  /* _MISRA_RULES */

#define my_flag 1u

uint32_t my_state(uint32_t state)
{
   return state & my_flag;/* o.k both unsigned */
}
```

Rules Descriptions

The following are brief explanations of how some of the MISRA-C rules are supported and interpreted in this CCES release due to the fact that some rules are handled in a nonstandard way, or some are not handled at all.

NOTE: Since the data types char, short and int are all represented as 32-bit integers on the SHARC architecture, MISRA rules relating to the size of variables may not be issued.

- *Rule 1.4 (required): The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.*

  The compiler and linker fully support this requirement.

- *Rule 1.5 (required): Floating-point implementations should comply with a defined floating-point standard.*

  Refer to Floating-Point Data Types.

- *Rule 2.4 (advisory): Sections of code should not be "commented out".*

  A diagnostic is reported if one of the following is encountered inside of a comment.

  `- character `{` or `}'`

  `- character `;' followed by a new-line character`

- *Rule 5.1 (required): Identifiers (internal and external) shall not rely on the significance of more than 31 characters.*

  This rule is only enforced when the `-misra-strict` compiler switch is used.

- *Rule 5.5 (advisory): No object or function identifier with static storage duration should be reused.*

  This rule is enforced by the compiler prelinker. The compiler generates `.misra` extension files that the prelinker uses to ensure that the same identifier is not used at file-scope within another module. This rule is not enforced if the `-misra-no-cross-module` compiler switch is used.

- *Rule 5.7 (advisory): No identifier shall be reused.*

  This rule is limited to a single source file. The rule is only enforced when the `-misra-strict` compiler switch is used.

- *Rule 6.3 (advisory): typedefs that indicate size and signedness should be used in place of basic types.*

  The `typedefs` for the basic types are provided by the system header files `<stdint.h>` and `<stdbool.h>`. The rule is only enforced when the `-misra-strict` compiler switch is used.

- *Rule 6.4 (advisory): Bit fields shall only be defined to be of type unsigned int or signed int.*

  The rule regarding the use of plain `int` is only enforced when the `-misra-strict` compiler switch is used.

- *Rule 8.1 (required): Functions shall have prototype declarations and the prototype shall be visible at both the function definition and the call.*

  For static and inline functions this rule is only enforced when the `-misra-strict` compiler switch is used.

- *Rule 8.5 (required): There shall be no definitions of objects or functions in a header file.*

  This rule does not apply to inline functions.

- *Rule 8.8 (required): An external object or function shall be declared in one and only one file.*

  This rule is enforced by the compiler prelinker. The compiler generates `.misra` extension files that the prelinker uses to ensure that the global is used in another file. The rule is not enforced if the `-misra-no-cross-module` switch is used.

- *Rule 8.10 (required): All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.*

  This rule is enforced by the compiler prelinker. The compiler generates `.misra` extension files that the prelinker uses to ensure that the global is used in another file. The rule is not enforced if the `-misra-no-cross-module` switch is used.

- *Rule 9.1 (required): All automatic variables shall have been assigned a value before being used.*

  The compiler attempts to detect some instances of violations of this rule at compile-time. There is additional code added at run-time to detect unassigned scalar variables. The additional Integral types with a size less than an int are not checked by the additional run-time code. This check also is available separately, via the `-rtcheck` and `-rtcheck-unassigned` switches. The run-time code is not added if `-misra-no-runtime` or `-no-rtcheck-unassigned` is used.

- *Rule 10.5 (required): If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.*

  When constant-expressions violate this rule, they are detected only when the `-misra-strict` compiler switch is used.

- *Rule 11.3 (advisory): A cast shall not be performed between a pointer type and an integral type.*

  The compiler always allows a constant of integral type to be cast to a pointer to a volatile type.

  ```
  volatile int32_t *n;
  n = (volatile int32_t *)10;
  ```

  There is only one case where this rule is not applied.

  ```
  int32_t *n;
  n = (int32_t *)10;
  ```

- *Rule 12.4 (required): The right-hand operand of a logical && or || operator shall not contain side-effects.*

  A function call used as the right-hand operand will not be faulted if it is declared with an associated `#pragma pure` directive.

- *Rule 12.7 (required): Bitwise operators shall not be applied to operands whose underlying type is signed.*

  The compiler will not enforce this rule if the two operands are constants.

- *Rule 12.8 (required): The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.*

If the right-hand operand is not a constant expression, the violation will be checked by additional run-time code when `-misra-no-runtime` is not enabled. If both operands are constants, the rule is only enforced when neither the `-misra-strict` compiler switch nor the `-no-rtcheck-shift-check` switch is used. This check is also available separately, via the `-rtcheck` and `-no-rtcheck-shift-check` switches.

- ***Rule 12.12 (required): The underlying bit representations of floating-point values shall not be used.***

  MISRA-C rules such as 11.4 prevent casting of bit-patterns to floating-point values. Hexadecimal floating-point constants are also not allowed when MISRA-C switches are used.

- ***Rule 13.2 (advisory): Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.*** The compiler treats variables which use the type `bool` (a typedef is declared in `<stdbool.h>`) as *Effectively Boolean* and will not raise an error when these are implicitly tested as zero, as follows:

```
bool b = 1;
if (b)
   ...;
```

- ***Rule 13.7 (required): Boolean operations whose results are invariant shall not be used.***

  The compiler does not detect cases where there is a reliance on more than one conditional statement. Constant expressions violating the rule are only detected when the `-misra-strict` compiler switch is used.

- ***Rule 16.2 (required): Functions shall not call themselves, either directly or indirectly.***

  A compile-time check is performed for a single file. Run-time code is added to ensure that functions do not call themselves directly or indirectly, but this code is not generated if the `-misra-no-runtime` compiler switch is used.

- ***Rule 16.4 (required): The identifiers used in the declaration and definition of a function shall be identical.***

  A declaration of a parameter name may have one leading underscore that the definition does not contain. This is to prevent name clashing. If the `-misra-strict` compiler switch is enabled, the underscore is significant and results in the violation of this rule.

- ***Rule 16.5 (required): Functions with no parameters shall be declared and defined with parameter type void.***

  Function `main` shall only be reported as violating this rule if the `-misra-strict` compiler switch is used.

- ***Rule 16.10 (required): If a function returns error information, then the error information shall be tested.***

  A function declared with return type `bool`, which is a `typedef` declared in header file `<stdbool.h>` will be faulted if the result of the call is not used.

- ***Rule 17.1 (required): Pointer arithmetic shall only be applied to pointers that address an array or array element.***

  Checking is performed at runtime. A run-time function looks at the value of the pointer and checks to see whether it violates this rule. This check is also available via the `-rtcheck` and `-rtcheck-arr-bnd` switches. It can be disabled via the `-no-rtcheck-arr-bnd` switch.

- ***Rule 17.2 (required): Pointer subtraction shall only be applied to pointers that address elements of the same array.*** Checking is performed at runtime. A run-time function looks at the value of the pointers and checks to see whether it violates this rule.
- ***Rule 17.3 (required): >, >=, <, <= shall not be applied to pointers that address elements of different arrays.***

  Checking is performed at runtime. A run-time function looks at the value of the pointers and checks to see whether it violates this rule.
- ***Rule 17.4 (required): Array indexing shall be the only allowed form of pointer arithmetic.***

  Checking is performed at runtime to ensure the object being indexed is an array. A run-time function looks at the value of the pointers and checks to see whether it violates this rule.

  All other forms of pointer arithmetic are reported at compile-time as violations of this rule.
- ***Rule 17.6 (required): The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.***

  Rule is not enforced under the following circumstances: if the address of a local variable is passed as a parameter to another function, the compiler cannot detect whether that address has been assigned to a global object.
- ***Rule 18.2 (required): An object shall not be assigned to an overlapping object.***

  The rule is not enforced by the compiler.
- ***Rule 18.3 (required): An area of memory shall not be reused for unrelated purposes.***

  The rule is not enforced by the compiler.
- ***Rule 19.4 (required): C macros shall only expand to a braced initializer, a constant, a string literal, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.***

  Use of `#pragma diag(suppress:misra_rule_19_4)` will suppress violations of this rule for any macro expansion during the scope of the suppression. If a macro is defined within the scope of the suppression, then the macro expansion will not be detected for violation of rule 19.4 even if the expansion point does not suppress the rule. See Diagnostic Control Pragmas.
- ***Rule 19.7 (advisory): A function shall be used in preference to a function-like macro.***

  The rule is only enforced when the compiler switch `-misra-strict` is used.
- ***Rule 19.15 (required): Precautions shall be taken in order to prevent the contents of a header file being included twice.***

  The compiler will report this violation if a header file is included more than once and does not prevent re-declarations of types, variables or functions.
- ***Rule 20.3 (required): The validity of values passed to library functions shall be checked.***

  This is not enforced by the compiler. The rule puts the responsibility on the programmer.
- ***Rule 20.4 (required): Dynamic heap memory allocation shall not be used.***

Prototype declarations for functions performing heap allocation should be declared with an associated `#pragma misra_func(heap)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- ***Rule 20.7 (required): The setjmp macro and longjmp function shall not be used.***

  Prototype declarations for these should be declared with an associated `#pragma misra_func(jmp)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- ***Rule 20.8 (required): The signal handling facilities of <signal.h> shall not be used.***

  Prototype declarations for functions in this header should be declared with an associated `#pragma misra_func(handler)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- ***Rule 20.9 (required): The input/output library <stdio.h> shall not be used.***

  Prototype declarations for functions in this header should be declared with an associated `#pragma misra_func(io)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- ***Rule 20.10 (required): The library functions atof, atoi and atol from library <stdlib.h> shall not be used.***

  Prototype declarations for these functions should be declared with an associated `#pragma misra_func(string_conv)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- ***Rule 20.11 (required): The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.***

  Prototype declarations for these functions should be declared with an associated `#pragma misra_func(system)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- ***Rule 20.12 (required): The time handling functions of library <time.h> shall not be used.***

  Prototype declarations for these functions should be declared with an associated `#pragma misra_func(time)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- ***Rule 21.1 (required): Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.***

  The compiler performs some static checks on uses of unassigned variables before conditional code and use of constant expressions. The compiler performs run-time checks for arithmetic errors, such as division by zero, array bound errors, unassigned variable checking and pointer dereferencing. Run-time checking has a negative effect on code performance. The `-misra-no-runtime` compiler switch turns off the run-time checking.

Run-Time Checking

The compiler provides support for detecting common programming mistakes, such as dereferencing a NULL pointer, or accessing an array beyond its bounds. The compiler does this by generating additional code to check for such conditions at runtime. Such code occupies space and incurs a performance penalty, so you should only use run-time checking when developing and debugging your application; products for release should always have run-time checking disabled.

The compiler's run-time checks are a subset of those enabled when MISRA-C run-time checking is active. For more information, see MISRA-C Compiler.

The following sections describe run-time checking in more detail:

- Enabling Run-Time Checking
- Supported Run-Time Checks
- Response When Problems Are Detected
- Limitations of Run-Time Checking

Enabling Run-Time Checking

Because of the associated overheads, run-time checking is disabled by default. You can enable run-time checking:

- By specifying command-line switches;
- Through the IDE, via run-time checking options under ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Run-time Checks***.

In both cases, you can enable all supported run-time checks, or just enable specific subsets.

Once run-time checking is enabled to some level, you can further turn that checking off and on again within your code, with pragmas. This allows you to narrow your focus down to particular functions, or to exclude certain functions from checking.

Command-Line Switches for Run-Time Checking

The following switches are used to turn run-time checking on:

- `-rtcheck`. Turns on all run-time checks.
- `-rtcheck-arr-bnd`. Turns on checking of array boundaries.
- `-rtcheck-div-zero`. Turns on checking for division by zero.
- `-rtcheck-heap`. Turns on checking of heap operations.
- `-rtcheck-null-ptr`. Turns on checking for NULL pointer dereferencing.
- `-rtcheck-shift-check`. Turns on checking of shift operations.
- `-rtcheck-stack`. Turns on checking for stack overflow.

- `-rtcheck-unassigned`. Turns on checking for use of variables before they've been assigned values.

The following switches are used to turn run-time checking off:

- `-no-rtcheck` . Turns off all run-time checks.
- `-no-rtcheck-arr-bnd`. Turns off checking of array boundaries.
- `-no-rtcheck-div-zero`. Turns off checking for division by zero.
- `-no-rtcheck-heap`. Turns off checking of heap operations.
- `-no-rtcheck-null-ptr`. Turns off checking for NULL pointer dereferencing.
- `-no-rtcheck-shift-check`. Turns off checking of shift operations.
- `-no-rtcheck-stack`. Turns off checking for stack overflow.
- `-rtcheck-unassigned`. Turns off checking for use of variables before they've been assigned values.

You can use combinations of these switches to enable the subset you require. For example, the following two sets of switches are equivalent:

- `-rtcheck -no-rtcheck-arr-bnd -no-rtcheck-div-zero\-no-rtcheck-heap -no-rtcheck-stack`
- `-rtcheck-null-ptr -rtcheck-shift-check -rtcheck-unassigned`

For more information, see `-rtcheck`.

Pragmas for Run-Time Checking

The following pragmas are used to enable and disable run-time checks.

- #pragma rtcheck(on). Turns on that subset of run-time checking that has been enabled by command-line switches.
- #pragma rtcheck(off). Turns off all run-time checking.

Note that these pragmas do not affect which run-time checks apply. Use command-line switches to select the appropriate checks, then use the pragmas to enable those checks during compilation of your functions of interest.

These pragmas cannot disable and re-enable heap-operation checking, or stack overflow detection. These checks are provided by linking in alternative library support, which apply to the whole application.

For more information, see Run-Time Checking Pragmas.

Supported Run-Time Checks

The following run-time checks are supported by the compiler:

Array Boundary Checks

When generating code to access arrays, the compiler generates additional code to see whether the location accessed falls within the boundaries of an array.

Division by Zero Checks

When generating code to perform an integer or floating-point division, the compiler generates additional code to check that the divisor is non-zero.

Heap Checks

The debugging version of the heap library checks for leaks, multiple frees of the same pointer, writes beyond the bounds of an allocation, and so on.

NULL Pointer Checks

When generating code to read the value pointed to by a pointer, the compiler generates additional code to verify that the pointer is not NULL.

Shift Checks

When generating code to shift a value X by some amount Y, the compiler generates additional code to check that:

- Y is not a negative value.
- Y is less than the number of bits required to represent X's type.

Stack Overflow Checks

When enabled, the run-time environment makes use of the SHARC processors circular-buffer mechanism for stack overflow detection: if the stack pointer (I7) advances beyond the limits defined by its corresponding base and length registers, a circular-buffer interrupt (CB7I) occurs and is trapped.

Unassigned Variable Checks

When generating code to read the value of a variable, the compiler generates additional code to make sure a value has previously been assigned to the variable.

Response When Problems Are Detected

In most cases, the additional code generated by the compiler includes code for emitting a diagnostic message to the `stderr` stream. This message is emitted when the run-time check finds a problem.

When stack overflow is detected, however, the generated code transfers control to the special label `__adi_stack_overflowed`, as emitting a diagnostic to the stderr stream would require additional stack space.

The IDE normally places a breakpoint on the `__adi_stack_overflowed` label. For more information, see Stack Overflow Detection.

The heap debugging library also provides support for logging problems to a file instead of reporting them immediately to the stderr stream. For more information, see Heap Debugging.

Limitations of Run-Time Checking

Besides the space/performance overheads incurred by the additional code, the following limitations apply to run-time checking:

- Compiled code only

  Because the run-time checks rely on additional code emitted during function compilation, the run-time checks can only apply to code compiled by the compiler, while run-time checks are enabled. Hand-written assembly or previously-compiled code cannot make benefit from run-time checking.

- No asm statements

  The compiler has no visibility into the contents of asm statements, so any actions carried out by asm statements will not be checked by any enabled run-time checking. For more information, see Inline Assembly Language Support Keyword (asm).

- Stdio support required

  Because the generated diagnostics are emitted to the stderr stream, run-time checking is only beneficial when the application supports the standard error stream, and the stream is attached to some suitable output device (such as the IDE console, which is the usual case when running an application within the debugger).

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. Most extensions are also available when compiling in C++ mode.

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

- Function Inlining
- Variable Argument Macros
- Restricted Pointers
- Variable-Length Array Support
- Non-Constant Initializer Support
- Designated Initializers
- Hexadecimal Floating-Point Numbers

- Declarations Mixed With Code
- Compound Literals
- C++ Style Comments
- Enumeration Constants That Are Not int Type
- Boolean Type

This section also contains information on other language extensions:

- The fract Native Fixed-Point Type
- Inline Assembly Language Support Keyword (asm)
- Dual Memory Support Keywords (pm dm)
- Memory Banks
- Placement Support Keyword (section)
- Placement of Compiler-Generated Code and Data
- Long Identifiers
- Preprocessor Generated Warnings
- Compiler Built-In Functions
- Pragmas
- GCC Compatibility Extensions
- Support for 40-Bit Arithmetic
- SIMD Support
- Accessing External Memory on ADSP-2126x and ADSP-2136x Processors

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 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.

The ***Keyword Extensions*** table provides a brief description of each keyword extension and directs you to sections of this chapter that document the extensions in more detail. The ***Operational Extensions*** table provides a brief description of each operational extension and directs you to sections that document these extensions in more detail.

Table 2-32: Keyword Extensions

Keyword Extensions	*Description*
`inline`	Directs the compiler to integrate the function code into the code of the calling function(s). For more information, see Function Inlining.
`asm()`	Places ADSP-21xxx/SCxxx assembly language instructions directly in your C/C++ program. For more information, see Inline Assembly Language Support Keyword (asm).
`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).
`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).
`section("string")`	Specifies thesection 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).
`bool`, `true`, `false`	A boolean type. For more information, see Boolean Type.
`restrict` keyword	Specifies restricted pointer features. For more information, see Restricted Pointers.
`fract`	Support for fixed-point arithmetic. For more information, see Using Native Fixed-Point Types.

Table 2-33: 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.
Long identifiers	Supports identifiers of up to 1022 characters in length. For more information, see Long Identifiers.
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.
Designated initializers	Support for designated initializers lets you specify elements of an aggregate initializer in arbitrary order. For more information, see Designated Initializers.
Compound literal expressions	Support for compound literals lets you create an aggregate array or structure value from component values within an expression. For more information, see Compound Literals.
Preprocessor-generated warnings	Lets you generate warning messages from the preprocessor. For more information, see Preprocessor Generated Warnings.

Table 2-33: Operational Extensions (Continued)

Operation Extensions	Description
C++ style comments	Allows for "//" C++ style comments in C programs. For more information, see C++ Style Comments.

Function Inlining

The `inline` keyword directs the compiler to integrate the code for the function you declare as `inline` into the code of its callers. Inline function support and the `inline` keyword is a standard feature of C++ and the ISO/IEC 9899:1999 C standard; the compiler provides this keyword as a C extension in C89 mode. For more information, see `-c89`.

Function inlining eliminates the function call overhead and increases the speed of your program's execution. Argument values that are constant and that have known values may permit simplifications at compilation time so that not all of the inline function's code need be included.

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

```
inline int max3 (int a, int b, int c) {
    return max (a, max(b, c));
}
```

The compiler can decide not to inline a particular function declared with the inline keyword, with a diagnostic remark cc1462 issued if the compiler chooses to do this. The diagnostic can be raised to a warning by use of the -Wwarn switch. For more information, see `-W{annotation|error|remark|suppress|warn}` *`number[, number ...]`*.

Function inlining can also occur by use of the `-Oa` (automatic function inlining) switch, which enables the inline expansion of C/C++ functions that are not necessarily declared inline in the source code. The amount of auto-inlining the compiler performs is controlled using the `-Ov` *`num`* (optimize for speed versus size) switch.

The compiler follows a specific order of precedence when determining whether a call can be inlined. The order is:

1. If the definition of the function is not available (for example, it is a call to an external function), the compiler cannot inline the call.
2. If the `-never-inline` switch is specified, the compiler does not inline the call. If the call is to a function that has #pragma always_inline specified (see Inline Control Pragmas), a warning will also be issued.
3. If the call is to a function that has `#pragma never_inline` specified, the call will not be inlined.
4. If the call is via a pointer-to-function, the call will not be inlined unless the compiler can prove that the pointer will always point to the same function definition.
5. If the call is to a function that has a variable number of arguments, the call will not be inlined.
6. If the module contains asm statements at global scope (outside function definitions), the call may not be inlined because the asm statement restricts the compiler's ability to reorder the resulting assembly output.

7. If the call is to a function that has #pragma always_inline specified, the call is inlined. If the call exceeds the current speed/space ratio limits, the compiler will issue a warning, but will still inline the call.
8. If the call is to a function that has the `inline` qualifier, or has `#pragma inline` specified, and the `-always-inline` switch has been used, the compiler inlines the call. If the call exceeds the current speed/space ratio limits, the compiler will issue a warning, but will still inline the call.
9. If the caller and callee are mapped to different code sections, the call will not be inlined unless the callee has the `inline` qualifier or has `#pragma inline` specified.
10. If the call is to a function that has the `inline` qualifier or has `#pragma inline` specified and optimization is enabled, the called function will be compared against the current speed/size ratio limits for code size and stack size. The calling function will also be examined against these limits. Depending on the limits and the relative sizes of the caller and callee, the inlining may be rejected.
11. If the call is to a function that does not have the `inline` qualifier or `#pragma inline`, and does not have `#pragma weak_entry`, then if the `-Oa` switch has been specified to enable automatic inlining, the called function will be considered as a possible candidate for inlining, according to the current speed/size ratio limits, as if the inline qualifier were present.

The compiler bases its code-related speed/size comparisons on the `-Ov` *`num`* num switch. When `-Ov` is in the range 1...100, the compiler performs a calculation upon the size of the generated code using the `-Ov` value, and this will determine whether the generated code is "too large" for inlining to occur. When `-Ov` has the value 1, only very small functions are considered small enough to inline; when `-Ov` has the value 100, larger functions are more likely to be considered suitable as well.

When `-Ov` has the value 0, the compiler is optimizing for space. The speed/space calculation will only accept a call for inlining if it appears that the inlining is likely to result in less code than the call itself would (although this is an approximation, since the inlining process is a high-level optimization process, before actual machine instructions have been selected).

Inlining and Optimization

The inlining process operates regardless of whether optimization has been selected (although if optimization is not enabled, then inlining will only happen when forced by #pragma always_inline or the `-always-inline` switch). The speed/size calculation still has an effect, although an optimized function is likely to have a different size from a non-optimized one, which is smaller (and therefore more likely to be inlined) and is dependent on the kind of optimization done.

A non-optimized function has loads and stores to temporary values which are optimized away in the optimized version, but an optimized function may have unrolled or vectorized loops with multiple variants, selected at run-time for the most efficient loop kernel, so an optimized function may run faster, but not be smaller.

Given that the optimization emphasis may be changed within a module – or even turned off completely – by the optimization pragmas, it is possible for either, both, or neither of the caller and callee to be optimized. The inlining process still operates, and is only affected by this in as far as the speed/size ratios of the resulting functions are concerned.

Inlining and Out-of-Line Copies

In C++ mode, the compiler conforms to the ISO/IEC:14882:2003 C++ standard and the ISO/IEC:14882:2011 C++ standard. In this mode the compiler will generate an out-of-line copy if the address of the function has been taken or the compiler has not been able to inline the call. Also, the out-of-line copy will have external linkage and be generated once by the prelinker if the function has been declared `extern inline`.

The following paragraphs describe the differences between C89 mode (`-c89`) and C99 mode (`-c99`). The use of inline is standard conforming in C99 mode and is an extension in C89 mode.

An inline function declared `static` behaves the same whether it is declared in C89 mode or C99 mode. No out-of-line copy is generated if all calls to the function are inlined and the address of the function is not taken.

An `inline` function declared with no storage class specifier behaves as follows. In C99 mode, if the address of the function is taken, then all calls not inlined within the current translation unit refer to an externally defined instance of the function, not the instance declared within this translation unit. In C99 mode, if the address of the function is not taken, it behaves as if the function has been declared with the keyword `static`. In C89 mode, it always behaves as if the function has been declared with the keyword `static`.

An `inline` function declared `extern` behaves differently in C99 mode (`-c99`) and C89 mode (`-c89`). In C99 mode, an external definition is always created. Therefore such a declaration should not be within a header file as this will result in a multiply defined symbol. In C89 mode, no out-of-line copy is created; therefore, if the address of the function is taken or a call is not inlined then an external reference is created and must be satisfied elsewhere.

Inlining and Global asm Statements

Inlining imposes a particular ordering on functions. If functions A and B are both marked as inline, and each calls the other, only one of the inline qualifiers can be followed. Depending on which the compiler chooses to apply, either A will be generated with inline versions of B, or B will be generated with inline versions of A. Either case may result in no out-of-line copy of the inlined function being generated. The compiler reorders the functions within a module to get the best inlining result. Functionally, the code is the same, but this affects the resulting assembly file.

When global `asm` statements are used with the module, between the function definitions, the compiler cannot do this reordering process, because the asm statement might be affecting the behavior of the assembly code that is generated from the following C function definitions. Because of this, global `asm` statements can greatly reduce the compiler's ability to inline a function call.

Inlining and Sections

Before inlining, the compiler checks any section directives or pragmas on the function definitions. For example,

```
section("secA") inline int add(int a, int b) { return a + b; }
section("secB") int times_two(int a) { return add(a, a); }
```

Since `add()` and `times_two()` are to be generated into different code sections, this call is ignored during the inlining process, so the call is not inlined. If the callee is marked with #pragma always_inline, however, or the `-always-inline` switch is in force, the compiler inlines the call despite the mismatch in sections.

Inlining and Run-Time Checking

When run-time checking is enabled, the compiler generates the additional code for the checks when the function is first defined. The implications for function inlining are as follows:

- When a function defined with run-time checking enabled is inlined into a function without run-time checking enabled, the inlined version still includes the run-time checks.
- When a function defined with run-time checking disabled is inlined into a function with run-time checking enabled, the inlined version does not acquire any run-time checks.

For more information, see Run-Time Checking.

Variable Argument Macros

This ISO/IEC 9899:1999 C standard feature is enabled as an extension in C89 mode and in C++ mode. The final parameter in a macro declaration may be an ellipsis (`...`) to indicate the parameter stands for a variable number of arguments. In the replacement text for the macro, the predefined name `__VA_ARGS__` represents the parameters that were supplied for the ellipsis in the macro invocation. At least one argument must be provided for the ellipsis, in an invocation.

For example:

```
#define tracec99(file,line,...) logmsg(file,line, __VA_ARGS__)
```

can be used with differing numbers of arguments: the following statements:

```
tracec99("a.c", 999, "one", "two", "three");
tracec99("a.c", 999, "one", "two");
tracec99("a.c", 999, "one");
tracec99("a.c", 999);
```

expand to the following code:

```
logmsg("a.c", 999, "one", "two", "three");
logmsg("a.c", 999, "one", "two");
logmsg("a.c", 999, "one");
logmsg("a.c", 999, );    // error - must provide an argument
```

NOTE: This variable argument macro syntax comes from the ISO/IEC 9899:1999 C standard. The compiler supports C99 variable argument macro formats in C89, C99 and C++ modes. See GCC Variable Argument Macros for additional information related to GNU compatibility support.

Restricted Pointers

The `restrict` keyword is a standard feature of the ISO/IEC 9899:1999 C standard. The keyword is available as an extension in C89 and C++ modes.

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, the `restrict` keyword is a way to identify that a

pointer does not create an alias. Also, two different restricted pointers cannot designate the same object, and therefore, they are not aliases.

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

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 a program 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. This ISO/IEC 9899:1999 standard feature is also allowed as an extension in C89 mode. For more information, see `-c89`. Variable-length arrays are not supported in C++ mode.

Unlike other automatic arrays, variable-length arrays are declared with a non-constant length. This means that the space is allocated when the array is declared, and deallocated when the brace-level is exited. The compiler does not allow jumping into the brace-level of the array, and produces a compilation 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:

```
void tester (int len, char data[len][len])
{
   /* code using data[][] */
}
```

The variable used for the array length must be in scope, and must have been previously declared.

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

more convenient than allocating it from a heap. Note, however, that variable-length arrays are allocated on the stack, which means that sufficient stack space must be available.

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 compiler does not require the elements of an aggregate initializer for an automatic variable to be constant expressions. This is a standard feature of the ISO/IEC 9899:1999 C standard and the ISO/IEC 14882:2003 C++ standard. The compiler supports it as an extension in C89 mode.

The following example shows an initializer with elements that vary at runtime.

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

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

Designated Initializers

This is a standard feature of the ISO/IEC 9899:1999 C standard. The compiler supports it as an extension in C89 and C++ modes, except for initializing arrays in C++11 mode.

This feature lets you specify the elements of an array or structure initializer in any order by specifying their designators-the array indices or structure field names to which they apply. All designators must be constant expressions, even in automatic arrays.

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

The following example shows equivalent array initializers-the first in C89 form (without using the extension) and the second in C99 form, using the designators. Note that the [INDEX] designator precedes the value being assigned to that element.

```
/* Example 1 C Array Initializer */
/* C89 array initializer (no designators) */
int a[6] = { 0, 0, 15, 0, 29, 0 };

/* Equivalent C99 array initializer (with designators) */
int a[6] = { [4] 29, [2] 15 };
```

You can combine this technique of designated elements with initialization of successive non-designated elements. The two instructions below are equivalent. Note that any non-designated initial value is assigned to the next consecutive element of the structure or array.

```
/* Example 2 Mixed Array Initializer */
/* C89 array initializer (no designators) */
int a[6] = { 0, v1, v2, 0, v4, 0 };

/* Equivalent C99 array initializer (with designators) */
int a[6] = { [1] v1, v2, [4] v4 };
```

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

```
/* Example 3 C Array Initializer With enum Type Indices */
/* C99 C array initializer (with designators) */
int whitespace[256] = {
   [' '] 1, ['\t'] 1, ['\v'] 1, ['\f'] 1, ['\n'] 1, ['\r'] 1
};

enum { e_ftp = 21, e_telnet = 23, e_smtp = 25, e_http = 80, e_nntp = 119 };
char *names[] = {
   [e_ftp] "ftp",
   [e_http] "http",
   [e_nntp] "nntp",
   [e_smtp] "smtp",
   [e_telnet] "telnet"
};
```

In a structure initializer, specify the name of the field to initialize with *fieldname*: before the element value. The C89 and C99 struct initializers in the example below are equivalent.

```
/* Example 4 struct Initializer */
/* C89 struct Initializer (no designators) */
struct point {int x, y;};
```

```
struct point p = {xvalue, yvalue};

/* Equivalent C99 struct Initializer (with designators) */
struct point {int x, y;};
struct point p = {y: yvalue, x: xvalue};
```

Hexadecimal Floating-Point Numbers

This is a standard feature of the ISO/IEC:9899 1999 C standard. The compiler supports this as an extension in C89 mode and in C++ mode.

Hexadecimal floating-point numbers have the following syntax.

```
hexadecimal-floating-constant:
   {0x|0X} hex-significand binary-exponent-part [ floating-suffix ]
hex-significand: hex-digits [ . [ hex-digits ]]
   binary-exponent-part: {p|P} [+|-] decimal-digits
floating-suffix: { f | l | F | L }
```

The `hex-significand` is interpreted as a hexadecimal rational number. The digit sequence in the exponent part is interpreted as a decimal integer. The 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`.

Declarations Mixed With Code

In C89 mode, the compiler accepts declarations placed in the middle of code. This allows the declaration of local variables to be placed at the point where they are required. Therefore, the declaration can be combined with initialization of the variable. This is a standard feature of the ISO/IEC 9899:1999 C standard and the ISO/IEC 14882:2003 C++ standard.

For example, in the following function the declaration of `d` is delayed until its initial value is available, so that no variable is uninitialized at any point in the function.

```
void func(Key k) {
   Node *p = list;
   while (p && p->key != k)
      p = p->next;
   if (!p)
      return;
   Data *d = p->data;
   while (*d)
```

```
    process(*d++);
}
```

Compound Literals

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

The following example shows an ISO/IEC 9899:1990 standard C `struct` usage, followed by an equivalent ISO/IEC 9899:1999 standard C code that has been simplified using a compound literal.

```
/* C89/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 C99 constructor struct */
struct foo make_foo(int x, char *s)
{
  return((struct foo) {x, {s[0], s[0] ? s[1] : '\0'}});
}
```

C++ Style Comments

The compiler accepts C++ 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;
```

and a C++ compiler and `cc21k` process as:

```
a = b;
```

Enumeration Constants That Are Not int Type

The compiler allows enumeration constants to be integer types other than `int`, such as `unsigned int`, `long long`, or `unsigned long long`. See Enumeration Type Implementation Details for more information.

Boolean Type

The compiler supports a Boolean data type `bool`, with values `true` and `false`. This is a standard feature of the ISO/IEC 14882:2003 C++ standard, and is available as a standard feature in the ISO/IEC 9899:1999 C standard when the `stdbool.h` header is included. It is supported as an extension in C89 mode, and as an extension in C99 mode when the `stdbool.h` header has not been included.

The `bool` keyword is a unique signed integral type. There are two built-in constants of this type: `true` and `false`. When converting a numeric or pointer value to `bool`, a zero value becomes `false,` and a non-zero value becomes `true`. A `bool` value may be converted to `int` by promotion, taking `true` to one and `false` to zero. A numeric or pointer value is converted automatically to `bool` when needed.

The fract Native Fixed-Point Type

The compiler has support for the native fixed-point type `fract`, as defined by Chapter 4 of the *Extensions to support embedded processors* ISO/IEC draft technical report TR 18037. This support is available for the C language only. A discussion of how to use this support is given in Using Native Fixed-Point Types.

Inline Assembly Language Support Keyword (asm)

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

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

NOTE: 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. It means that the compiler cannot apply any enabled workarounds for silicon errata that may be triggered either by the contents of the asm construct, or by the sequence of instructions formed by the `asm()` construct and the surrounding code produced by the compiler.

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

`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.

In addition, 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 of:

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

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

NOTE: The compiler generates a label before and after inline assembly instructions when generating debug code (the `-g` switch). These labels are used to generate the debug line information used by the debugger. If the inline assembler inserts conditionally assembled code, an undefined symbol error is likely to occur at link time. 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 thus 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.

- asm() Construct Syntax
- Assembly Construct Operand Description
- Using long long Types in asm Constraints
- Assembly Constructs With Multiple Instructions
- Assembly Construct Reordering and Optimization
- Assembly Constructs With Input and Output Operands
- Assembly Constructs With Compile-Time Constants
- Assembly Constructs and Flow Control
- Guidelines on the Use of asm() Statements

asm() Construct Syntax

Use `asm()` constructs to specify the operands of the assembly instruction using C/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 [volatile] (
template
  [:[constraint(output operand)[,constraint(output operand)…]]
    [:[constraint(input operand)[,constraint(input operand)…]]
```

```
      [:clobber string]]]
);
```

The syntax elements are:

- *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. Each instruction should end 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.

- *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.

- *output operand*

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

- *input operand*

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

- *clobber string*

  The clobber string notifies the compiler that a list of registers are overwritten by the assembly instructions. Enclose each name within double quotes, and separate each quoted register name with a comma. The input and output operands are guaranteed not to use any of the clobbered registers, so you can read and write the clobbered registers as often as you like. See the ***Register Names for asm() Constructs*** table in Assembly Construct Operand Description for the list of individual registers. See the ***Clobbered Register Sets*** table (in #pragma regs_clobbered *string*) for the list of register sets.

It is vital that any register overwritten by an assembly instruction and not allocated by the constraints is included in the clobber list. The list must include memory if an assembly instruction accesses memory.

asm() Construct Syntax Rules

These rules apply to assembly construct template syntax.

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

    - The template from the first output operand
    - The last output operand from the first input operand
    - The last input operand from the clobbered registers
- A space must be 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).

NOTE: 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:

- 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 `d` 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.

The ***asm() Operand Constraints*** table describes the correspondence between constraint letters and register classes.

NOTE: The use of any letter not listed in the ***asm() Operand Constraints*** table results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

Table 2-34: asm() Operand Constraints

Constraint	*Register Type*	*Register*
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
I	64-bit R register pair	R0 - R15 (For more information, see Using long long Types in asm Constraints.)
j	DAG1 L registers	L0 - L7
k	Q1 R registers	R0 - R3
l	Q4 R registers	R12 - R15
r	General registers	R0 - R15
u	User registers	USTAT1 - USTAT4
w	DAG1 I registers	I0 - I7
x	DAG1 M registers	M0 - M7
y	DAG2 I registers	I8 - I15
z	DAG2 M registers	M8 - M15
n	None. For more information, see Assembly Constructs With Compile-Time Constants.	
=&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.	
&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.	

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 (in the ***asm() Operand Constraints*** table) describe which registers are modified by an assembly language instruction. The "=" in =constraint indicates that the operand is an output; all output operand constraints must use =. Operands that are input-outputs must use "+". (See below.)
- The compiler may allocate an output operand in the same register as an unrelated input operand, unless the output or input operand has the & constraint modifier. This 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

The ***asm() Operand Constraints*** table lists the registers that may be allocated for each register constraint letter. The use of any letter not listed in the ***Constraint*** column of the table results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter. The ***Register Names for asm() Constructs*** table lists the registers that may be named as part of the clobber list.

Table 2-35: 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
"f0", "f1", "f2", "f3", "f4", "f5", "f6", "f7", "f8", "f9", "f10", "f11", "f12", "f13", "f14", "f15"	Floating-point 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

Table 2-35: Register Names for asm() Constructs (Continued)

Clobber String	*Meaning*
"lcntr"	Loop counter register
"px"	PX register
"ustat1", "ustat2", "ustat3", "ustat4"	User-defined status registers
"s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9", "s10", "s11", "s12", "s13", "s14", "s15"	Shadow data registers
"msf", "msb"	Shadow multiplier result registers
"sacc", "smcc", "sscc", "sbtf"	Shadow condition registers
"memory"	Unspecified memory locations

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 the ***asm() Operand Constraints*** table.

For example,

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

loads `x` into `r14`, loads `y` into `r15`, executes the operation, and then stores the total from `r13` back into `result`.

NOTE: 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.

Using long long Types in asm Constraints

It is possible to use an `asm()` constraint to specify a `long long` value, in which case the compiler will claim a valid register pair. The syntax for operands within the template is extended to allow the suffix "`H`" for the high-numbered register of the register pair, and the suffix "`L`" for the low-numbered register of the pair. A `long long` type is represented by the constraint letter "`I`". Note that when using a char-size of 8 bits, the high-numbered register contains the least-significant bits of the `long long` value and the low-numbered register contains the most-significant bits, while when using a char-size of 32 bits, the high-numbered register contains the most-significant bits of the `long long` value and the low-numbered register contains the least-significant bits.

For example,

```
long long int res;
int main(void) {
   long long result64, x64 = 123;
   asm(
```

```
     "%0H = %1H; %0L = %1L;" :
     "=I" (result64) :
     "I" (x64)
     );
   res = result64;
}
```

In this example, the template is "`%0H=%1H; %0L=%1L;`". When compiling with a char-size of 32 bits, the `%0H` is replaced with the register containing the least-significant 32 bits of operand zero (`result64`), and `%0L` is replaced with the register containing the most-significant 32 bits of operand zero (`result64`). Similarly, `%1H` and `%1L` are replaced with the registers containing the least-significant 32 bits and most-significant 32 bits, respectively, of operand one (x64).

Assembly Constructs With Multiple Instructions

There can be many assembly instructions in one template. Normal rules for line-breaking apply. In particular, the statement may spread over multiple lines. You are recommended not to split a string over more than one line, but to use the C language's string concatenation feature. If you are placing the inline assembly statement in a preprocessor macro, see Compound Macros

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", “astat”); /* 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) : “astat”);
```

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 reorder them with respect to other statements, where there is no visible data dependency. 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 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/C ++ statement.

Assembly Constructs With Input and Output Operands

When an asm construct has both inputs and outputs, there are two aspects to consider:

- Whether a value read from an input variable will be written back to the same variable or a different variable on output.
- Whether the input and output values will reside in the same register or different registers.

The most common case is when both input and output variables and input and output registers are different. In this case, the asm construct reads from one variable into a register, performs an operation which leaves the result in a different register, and writes that result from the register into a different output variable:

```
asm("%0 = %1;" : "=d" (newptr) : "d" (oldptr));
```

When the input and output variables are the same, it is usual that the input and output registers are also the same. In this case, you use the "+" constraint:

```
asm("%0 += 4;" : "+d" (sameptr));
```

When the input and output variables are different, but the input and output registers have to be the same (usually because of requirements of the assembly instructions), you indicate this to the compiler by using a different syntax for the input's constraint. Instead of specifying the register or class to be used, you specify the output to which the input must be matched.

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
```

This specifies that the input `oldptr` has 0 (zero) as its constraint string, which means it must be assigned the same register as %0 (`newptr`).

Assembly Constructs With Compile-Time Constants

The `n` input constraint informs the compiler that the corresponding input operand should not have its value loaded into a register. Instead, the compiler is to evaluate the operand, and then insert the operand's value into the assembly command as a literal numeric value. The operand must be a compile-time constant expression. For example,

```
int r; int arr[100];
asm("%0 = %1;" : "=d" (r) : "d" (sizeof(arr)));  // "d" constraint
```

produces code similar to:

```
R0 = 100 (X);   // compiler loads value into register
R1 = R0;        // compiler replaces %1 with register
```

whereas:

```
int r; int arr[100];
asm("%0 = %1;" : "=d" (r) : "n" (sizeof(arr)));  // "n" constraint
```

produces code similar to:

```
R1 = 100;       // compiler replaces %1 with value
```

If the expression is not a compile-time constant, the compiler gives an error:

```
int r; int arr[100];
asm("%0 = %1;" : "=d" (r) : "n" (arr));  // error: operand
                                         // for "n" constraint
                                         // must be a compile-time constant
```

Assembly Constructs and Flow Control

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

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

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

Guidelines 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 <def21160.h>

test = sysreg_bit_tst(sysreg_MODE1, IRPTEN);
```

Refer to Access to System Registers for more information.

Accessing Memory-Mapped Registers (MMRs)

MMRs can be accessed using the macros in the `Cdef*.h` files (for example, `Cdef21160.h`) that are supplied with CCES.

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 <Cdef21160.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 and ADSP-SCxxx 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;
/* declares xp as a pointer in Program Memory,
   that points to a pointer in Data Memory,
   which in turn points to a float back 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

The compiler allows a value of type pointer to `pm` to be assigned to a variable of type pointer to `dm`. Moreover, when the `-compatible-pm-dm` switch is used, the compiler also allows a value of type pointer to `dm` to be assigned to a variable of type pointer to `pm`. The same rules also apply when passing arguments to functions that take pointer-typed parameters.

Care must be taken when such coversions are performed. The compiler assumes that a variable of type pointer to dm and a variable of type pointer to pm can never point to the same object. It uses this assumption to generate more efficient code, by resolving the potential alias between two pointers. If you use values of both type pointer to pm and pointer to dm to reference the same underlying object, incorrect code may be generated, as the compiler is free to reorder any accesses to the object that use the different pointer types.

For example, suppose you have two stores in C and the pointers contain the same address:

```
*pm_ptr = v1;
*dm_ptr = v2;
```

In this case, the compiler may perform the two accesses in either order. This could result in the DM v2 store occurring before the PM v1 store and not as in the source if the pointers were the same address. While this is a contrived example, it is worth considering if mixing pm and dm pointers.

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 */
/* when -compatible-pm-dm isn't used */
int pm *x;
int dm *y;
int dm a;
x = y;         /* Compiler will flag error cc0513 */
x = &a;        /* Compiler will flag error cc0513 */
```

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

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

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

```
                    /* which resides in dm     */

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

Memory Keywords and Function Arguments

The compiler checks calls to prototyped functions for memory space specifications consistent with the function prototype. The following 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 dm 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 code segment guarded by `EASILY_CHANGED` lets you redefine the memory space specifications by redefining the macros `SPACE1` and `SPACE2`, and making it easy to redefine the memory space specifications at compile-time.

```
/* Dual Memory Support Keywords & Macros */

#ifdef EASILY_CHANGED
   /* pm and dm can be easily changed at compile-time. */
   #define SPACE1 pm
   #define SPACE2 dm
   char SPACE1 * foo (char SPACE2 *);
   char SPACE1 * x;
   char SPACE2 y;
   x = foo(&y);
```

```
#else
   /* not so easily changed. */
   char pm * foo (char dm *);
   char pm * x;
   char dm y;
   x = foo(&y);
#endif
```

Memory Banks

By default, the compiler assumes that all memory may be accessed with equal performance, but this is not always the case: some parts of your application may be in faster internal memory, and other parts in slower, external memory. The compiler supports the concept of memory banks to group code or data with equivalent performance characteristics. By providing this information to the compiler, you can improve the performance of your application.

Memory Banks Versus Sections

Note that memory banks are different from sections:

- Section is a "hard" directive, using a name that is meaningful to the linker. If the `.ldf` file does not map the named section, a linker error occurs.
- A memory bank is a "soft" informational characterization, using a name that is not visible to the linker. The compiler uses optimization to take advantage of the bank's performance characteristics. However, if the `.ldf` file maps the code or data to memory that performs differently, the application still functions (albeit with a possible reduction in performance).

Pragmas and Qualifiers

Memory banks may be referenced through both memory bank pragmas and memory bank qualifiers:

- Use memory bank pragmas to specify the memory banks used by all the code or data of a function. For example:

  ```
  #pragma data_bank(bank_external)
  int *getptr(void) { return ptr2; }
  ```

- Use memory bank qualifiers to specify the memory bank referenced by individual variables. For example:

  ```
  int bank("bank_internal") *ptr1;
  int bank("bank_external") *ptr2;
  ```

Memory Bank Selection

The compiler applies the following process for determine which bank is being referenced.

Memory Banks for Code

The compiler uses the following process for deducing the memory bank which contains instructions:

1. If the function is immediately preceded by `#pragma code_bank(bank)`, then the function's instructions are considered to reside in memory bank `bank`.

2. If the function is immediately preceded by `#pragma code_bank` or `#pragma code_bank()`, then the function's instructions are not considered to reside in any defined memory bank.
3. Otherwise, if `#pragma default_code_bank(defbank)` has been used in the compilation unit prior to the definition of the function, the function's instructions are considered to reside in memory bank `defbank`.
4. Otherwise, the function's instructions are not considered to reside in any defined memory bank.

For more information, see #pragma code_bank(*bankname*).

Memory Banks for Data

The compiler uses the following process for deducing which memory bank contains variables that are auto storage class:

1. If the variable declaration includes a memory bank qualifier, for example,

```
int bank("bank") x;
```

   then the variable will be considered to reside in bank `bank`.
2. Otherwise, if the function is immediately preceded by `#pragma stack_bank(bank)`, then the variable is considered to reside in memory bank `bank`.
3. Otherwise, if the function is immediately preceded by `#pragma stack_bank` or `#pragma stack_bank()`, then the variable is not considered to reside in any memory bank.
4. Otherwise, if `#pragma default_stack_bank(defbank)` has been used in the compilation unit prior to the definition of the function, the variable is considered to reside in memory bank `defbank`.
5. Otherwise, the variable is not considered to reside in any defined memory bank.

For more information, see #pragma stack_bank(*bankname*).

The compiler uses the following process for selecting the memory bank to contain `static` variables defined within a function:

1. If the variable declaration includes a memory bank qualifier, for example,

```
static int bank("bank") x;
```

   then the variable will be considered to reside in bank `bank`.
2. Otherwise, if the function is immediately preceded by `#pragma data_bank(bank)`, then the variable is considered to reside in memory bank `bank`.
3. Otherwise, if the function is immediately preceded by `#pragma data_bank` or `#pragma data_bank()`, then the variable is not considered to reside in any memory bank.
4. Otherwise, if `#pragma default_data_bank(defbank)` has been used in the compilation unit prior to the definition of the function, the variable is considered to reside in memory bank `defbank`.
5. Otherwise, the variable is not considered to reside in any defined memory bank.

For more information, see #pragma data_bank(*bankname*).

The compiler uses the following process for selecting the memory bank to contain variables defined at global scope:

1. If the variable declaration includes a memory bank qualifier, for example,

   ```
   int bank("bank") x;
   ```

   then the variable will be considered to reside in bank `bank`.

2. Otherwise, if `#pragma default_data_bank(defbank)` has been used in the compilation unit prior to the definition of the variable, the variable is considered to reside in memory bank `defbank`.

3. Otherwise, the variable is not considered to reside in any defined memory bank.

The identified memory bank is used for pointer dereferences. For example:

```
#pragma data_bank(bank_external)
int f(int *a, int *b) {
   return *a + *b;  // *a and *b both considered to be
}                   // loads from "bank_external"
```

For more information, see #pragma default_code_bank(*bankname*).

Performance Characteristics

You can specify the performance characteristics of a memory bank. This will allow the compiler to generate optimal code when accessing the bank. You can specify the following characteristics:

- Cycles required to read the memory bank. Use #pragma bank_read_cycles(*bankname, cycles[, bits]*) to specify this characteristic.
- Cycles required to write the memory bank. Use #pragma bank_write_cycles(*bankname, cycles[, bits]*) to specify this characteristic.
- The maximum bit width supported by accesses to the memory bank. Use #pragma bank_maximum_width(*bankname, width*) to specify this characteristic.

Memory Bank Kinds

Each memory bank has a defined kind. The memory bank kinds supported on SHARC processors are listed in the ***Memory Bank Kinds*** table. Not all kinds are available on all processors.

Table 2-36: Memory Bank Kinds

Kind	*Description*
`internal`	Corresponds to internal memory
`external`	Corresponds to memory that is external to the processor.

Predefined Banks

The compiler predefines a memory bank for each supported memory bank kind, using the same name but with a `bank_` prefix. For example, the following uses the internal and external memory banks:

```
#pragma code_bank("bank_external")
int next_counter(void) {
   static int bank("bank_internal") counter;
   return counter++;
}
```

These predefined memory banks have predefined performance characteristics, such as read and write cycle counts, that are appropriate for the kind of memory. You can override these performance characteristics via pragmas. For more information, see Memory Bank Pragmas.

The memory bank kinds are listed in the ***Memory Bank Kinds*** table (see Memory Bank Pragmas).

Defining Additional Banks

New memory banks are defined when first used, whether this happens in a memory bank pragma, or in a memory bank qualifier. When created, memory banks have kind `internal`, unless otherwise specified by `#pragma memory_bank_kind`.

The compiler does not attach any significance to the name of any new memory banks you create.

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 Section Usage.

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 definition of a static variable that is placed in the section called `bingo`.

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

The `section()` keyword has the limitation that section initialization qualifiers cannot be used within the section name string. The compiler may generate labels containing this string, which will result in assembly syntax errors. Additionally, the keyword is not compatible with any pragmas that precede the object or function. For finer control over section placement and compatibility with other pragmas, use `#pragma section`.

Refer to #pragma section/#pragma default_section for more information.

NOTE: Note that section has replaced the `segment` keyword in earlier releases of the compiler. Although the `segment()` keyword is supported by the compiler of the current release, we recommend that you revise legacy code.

Placement of Compiler-Generated Code and Data

If the `section()` keyword (Placement Support Keyword (section)) is not used, the compiler emits code and data into default sections. The `-section` switch (`-section`*`id=section_name[, id=section_name...]`*) can be used to specify alternatives for these defaults on the command-line, and the `default_section` pragma (#pragma section/#pragma default_section) can be used to specify alternatives for some of them within the source file.

In addition, when using certain features of C/C++, the compiler may be required to produce internal data structures. The `-section` switch and the `default_section` pragma allow you to override the default location where the data would be placed. For example,

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

When both `-section` switches and `default_section` pragmas are used, the default_section pragmas take priority.

Long Identifiers

The compiler supports C identifiers of up to 1022 characters in length; C++ identifiers typically have a slightly shorter limit, as the limit applies to the identifier after *name mangling* is used to transform it into a suitable symbol for linking, and for C++, some of the symbol space is required to represent the identifier's type.

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.

Compiler Built-In Functions

The compiler supports built-in functions (sometimes called intrinsics) 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 the C standard reserves identifiers beginning with double underlines (__), so these names do not conflict with user program identifiers.

This section describes:

- builtins.h

- Access to System Registers
- Circular Buffer Built-In Functions
- Compiler Performance Built-In Functions
- Floating-Point Built-in Functions
- Fractional Built-In Functions
- Multiplier Built-In Functions
- Exclusive Transaction Built-In Functions
- Multiplier Built-In Functions

The `cc21k` compiler provides built-in versions of some C library functions as described in ***Using Compiler Built-In C Library Functions*** of the *C/C++ Library Manual for SHARC Processors*.

builtins.h

The `builtins.h` header file defines prototypes for all built-in functions supported by the compiler; include this header file in any module that invokes a built-in function.

The header file also defines short names for each built-in function: for each built-in function `__builtin_func()`, the header file defines the short name *`func()`*. These short names can be disabled selectively or *en masse*, by defining macros prior to include the header file. The ***Macros controlling builtins.h*** table lists these macros.

Table 2-37: Macros controlling builtins.h

Macro Name	*Effect*
`__NO_SHORTNAMES`	If defined, prevents any short names from being defined.
`__SPECIFIC_NAMES`	If defined, short name *`func`* will only be defined if corresponding macro `__ENABLE_FUNC` is defined.
`__ENABLE_FUNC`	Causes short name *`func`* to be defined, if `__SPECIFIC_NAMES` is also defined.
`__DISABLE_FUNC`	Prevents short name *`func`* from being defined.
`__DEFINED_FUNC`	Multiple-inclusion guard. The header file defines this macro when it defines short name *`func`*, but will not define short name *`func`* if this macro is already defined.

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/SCxxx 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.

The functions are:

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

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

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

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

- ```
  void sysreg_write_nop (const int SR_number, const int new_value);
  ```

  The `sysreg_write_nop` function stores the specified value in the nominated system register, but also places a `NOP;` after the instruction.

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

  The `sysreg_bit_clr` function 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);
  ```

  The `sysreg_bit_clr_nop` function clears 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);
  ```

  The `sysreg_bit_set` function 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);
  ```

  The `sysreg_bit_set_nop` function sets all the bits of the nominated system register that are also 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);
  ```

  The `sysreg_bit_tgl` function 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);
  ```

  The `sysreg_bit_tgl_nop` function 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);
  ```

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

- ```
  int sysreg_tst (const int SR_number, const int value);
  ```
  ```

If the contents of the nominated system register are equal to the supplied value, the `sysreg_tst` function returns a non-zero value.

NOTE: The `SR_number` parameters must be compile-time constants; `sysreg.h` defines suitable macros. 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/SCxxx processors, use the following system registers:

```
sysreg_IMASK                  sysreg_IMASKPcodeblock
codeblock  sysreg_ASTAT      sysreg_STKYcodeblock
codeblock  sysreg_USTAT1     sysreg_USTAT2codeblock
codeblock  sysreg_USTAT3     sysreg_USTAT4codeblock
codeblock  sysreg_MODE1      sysreg_MODE2codeblock
codeblock  sysreg_IRPTL      sysreg_ASTATXcodeblock
codeblock  sysreg_STKYY      sysreg_MMASKcodeblock
codeblock  sysreg_ASTATY     sysreg_FLAGScodeblock
```

On the ADSP-211xx/212xx/213xx/214xx processors, the following system register is also available:

```
sysreg_LIRPTL
```

On the ADSP-215xx/SC5xx processors, the following system register is also available:

```
sysreg_MODE1STK
```

Header files specific to each processor provide symbolic names for the individual bits in the system registers of the processor; for example, `def21160.h` (for the ADSP-21160 processor) and `def21469.h` (for the ADSP-21469 processor). Including the header `platform_include.h` automatically includes the `def21xxx.h` header for the processor for which the application is being compiled.

Circular Buffer Built-In Functions

The C/C++ compiler provides built-in functions that use the processor's circular buffer mechanisms. These functions provide automatic circular buffer generation, circular indexing, and circular pointer references.

Automatic Circular Buffer Generation

If optimization is enabled, the compiler automatically attempts to use circular buffer mechanisms where appropriate. For example,

```
void func(int *array, int n, int incr)
{
   int i;
   for (i = 0; i < n; i++)
      array [ i % 10 ] += incr;
}
```

The compiler recognizes that the `[i % 10 ]` expression is a circular reference, and uses a circular buffer if possible. There are cases where the compiler is unable to verify that the memory access is always within the bounds of the buffer. The compiler is conservative in such cases, and does not generate circular buffer accesses.

The compiler can be instructed to still generate circular buffer accesses even in such cases, by specifying the -force-circbuf switch.

For more information, see -force-circbuf.

Circular Buffer Increment of an Index

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

```
ptrdiff_t circindex(ptrdiff_t ptr, ptrdiff_t incr, size_t len);
```

The equivalent operation is:

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

NOTE: Note that, for correct operation, the increment should not exceed the buffer length.

Circular Buffer Increment of a Pointer

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

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

The equivalent operation is:

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

NOTE: Note that, for correct operation, the increment should not exceed the buffer length.

For more information on `circindex` and `circptr` library functions, refer to the *C/C++ Library Manual for SHARC Processors*.

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 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

The compiler performance built-in functions do not have any effect on the functional behavior of compiled code. Instead, they provide the compiler with additional information about the code being compiled, allowing the compiler to generate more efficient code. The facilities are:

- Expected Behavior

- Known Values

Expected Behavior

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

#include <processor_include.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 most efficient 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 be most efficient for 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.

Known Values

The `__builtin_assert()` function provides the compiler with information about the values of variables which it may not be able to deduce from the context. For example, consider the code

```
int example(int value, int loop_count)
{
   int r = 0;
   int i;
   for (i = 0; i < loop_count; i++) {
      r += value;
   }
   return r;
}
```

The compiler has no way of knowing what values may be passed in to the function. If you know that the loop count will always be greater than four, you can allow the optimizer to make use of that knowledge using `__builtin_assert()`:

```
int example(int value, int loop_count)
{
   int r = 0;
   int i;
   __builtin_assert(loop_count > 4);
   for (i = 0; i < loop_count; i++) {
      r += value;
   }
   return r;
}
```

The optimizer can now omit the jump over the loop body it would otherwise have to emit to cover `loop_count == 0`. In more complicated code, further optimizations may be possible when bounds for variables are known.

Integral Built-in Functions

The compiler provides the following functions for operations on integral types, declared in `builtins.h`:

- `int avg(int x, int y);`

  Adds *x* and *y*, and divides the result by 2.

- `int clip(int val, int range);`

If `val` is greater than `|range|`, the function returns `|range|`, else if `val` is less than `-|range|` the function returns `-|range|`, else `val` is returned unchanged. This built-in function generates the `clip val by range` instruction.

Floating-Point Built-in Functions

The compiler provides the following functions for operations on floating-point types, declared in `builtins.h`:

- `float frecipsf(float val);`

  Creates a seed value for computing `1/val`, the reciprocal of `val`. This built-in function generates the `recips` instruction.

- `float frsqrtsf(float val);`

  Creates a seed value for computing the reciprocal square root of val. This built-in function generates the `rsqrts` instruction.

- `float fscalbf(float val, int amount);`

  Scales the exponent of `val` by adding `amount` to the exponent. This built-in function generates the `scalb val by amount` instruction.

- `float faddabsf(float x, float y);`

  Adds *x* and *y*, then returns the absolute value of the sum.

- `float favgf(float x, float y);`

  Adds *x* and *y*, and divides the result by 2.

- `float fsubabsf(float x, float y);`

  Subtracts *y* from *x*, then returns the absolute value of the resulting amount.

- `float faddsubf(float x, float y, float *z);`

  Subtracts *y* from *x*, storing the result in the location pointed to by `z`, then returns the result of adding `x` to `y`.

- `float fsignf(float x, float y);`

  Returns *x* if *y* is non-negative, and *-x* if *y* is negative. This built-in function generates the `copysign` instruction.

- `float fclipf(float val, float range);`

  If `val` is greater than `|range|`, the function returns `|range|`, else if `val` is less than `-|range|` the function returns `-|range|`, else `val` is returned unchanged. This built-in function generates the `clip val by range` instruction.

- `int conv_fix(float val);`

  Converts `val` to an integer. This built-in function generates the `fix` instruction, so rounding is affected by the `MODE1` register.

- `int conv_fix_by(float val, int amount);`

  Scales *val* by adding *amount* to *val* exponent, then converts the result to an integer. This built-in function generates the `fix val by amount` instruction, so rounding is affected by the `MODE1` register.

- `float conv_float_by(int val, int amount);`

  Converts *val* to a floating-point value, then scales that value by adding *amount* to the value's exponent. This built-in function generates the `float val by amount` instruction, so rounding is affected by the MODE1 register.

- `int conv_trunc_by(float val, int amount);`

  Scales *val* by adding *amount* to *val*'s exponent, then converts the result to an integer. This built-in function generates the `trunc val by amount` instruction; rounding always truncates towards zero.

- `int logbf(float val);`

  Returns the unbiased exponent of `val`. This built-in function generates the `logb` instruction.

- `int mantf(float val);`

  Returns the mantissa of `val` with the hidden bit reinstated. This built-in function generates the `mant` instruction.

- `int fpack(float val);`

  Returns an integer representing `val` in 16-bit floating-point format. This built-in function generates the `fpack` instruction.

- `float funpack(int val);`

  Returns the float value corresponding to the 16-bit floating-point value represented in `val`. This built-in function generates the `funpack` instruction.

Fractional Built-In Functions

The SHARC compiler provides a set of fractional built-in functions, declared in `builtins.h`. These built-in functions are:

- `float conv_RtoF(int __a);`

  Converts a fractional value to floating point representation. This function is implemented by a `float` instruction. Conversion from a fractional value to a floating-point value may result in some precision loss.

  An alternative way to generate the same code is to use the `fract` native fixed-point type. In this case, we can cast the `fract`-typed argument to `float` type without the need to use a built-in function. For more information, refer to Using Native Fixed-Point Types.

- `int conv_FtoR(float __a);`

Converts a floating point value to a fractional representation. This function is implemented by a `fix` instruction and does not saturate. Conversion of a floating point value that cannot be represented as a fractional value will return 0xFFFFFFFF.

Similar functionality is provided by the `fract` native fixed-point type. In this case, we can simply cast the `float`-typed argument to `fract` type without the need to use a built-in function. The behavior of this cast differs from the built-in, in that it does not depend on the rounding mode specified in the `MODE1` register. For more information, refer to Using Native Fixed-Point Types.

- `int RxR(int __a, int __b);`

  Multiplies two fractional values, returning a fractional value. This function is implemented by a multiply instruction followed by a `sat` instruction. The function will saturate. The operation `(-1)*(-1)` will return `0x7FFFFFFF`.

  An alternative way to generate the same code is to use the `fract` native fixed-point type. In this case, we multiply two `fract`-typed arguments directly using the unbiased rounding mode, without the need to use a built-in function. For more information, see Using Native Fixed-Point Types.

- `int RxItoI(int __a, int __b);`

  Multiplies a fractional value with an integral value, returning an integral value. This function is implemented as a multiply instruction followed by a `sat` instruction.

  Similar functionality is provided by the `fract` native fixed-point type through use of the `mulir` function (see mulifx). The behavior of this function differs from the built-in, in that it rounds towards zero and does not saturate.

- `int RxItoR(int __a, int __b);`

  Multiplies a fractional value with an integral value, returning a fractional value. This function is implemented by a multiply instruction followed by a `sat` instruction. This function will saturate. Any negative number that cannot be represented by fract will return `0x80000000`, and any positive number that cannot be represented will return `0x7FFFFFFF`.

  An alternative way to generate the same code is to use the `fract` native fixed-point type. In this case, we multiply a `fract`-typed and an `int`-typed argument directly without the need to use a built-in function. For more information, see Using Native Fixed-Point Types.

- `int sat_add(int __a, int __b);`

  Adds two fractional values, returning a fractional value. This function will saturate. Any negative number that cannot be represented by fract will return `0x80000000`, and any positive number that cannot be represented will return `0x7FFFFFFF`.

  An alternative way to generate the same code is to use the `fract` native fixed-point type. In this case, we add two `fract`-typed values directly without the need to use a built-in function. For more information, see Using Native Fixed-Point Types.

- `int sat_sub(int __a, int __b);`

Subtracts second fractional value argument from the first, returning a fractional value. This function will saturate. Any negative number that cannot be represented by fract will return `0x80000000`, and any positive number that cannot be represented will return `0x7FFFFFFF`.

An alternative way to generate the same code is to use the `fract` native fixed-point type. In this case, we subtract two `fract`-typed values directly without the need to use a built-in function. For more information, see Using Native Fixed-Point Types.

- `unsigned int leftz(unsigned int __a);`

  Returns the number of leading zeros in operand `__a`. This built-in function generates the `leftz` instruction.

- `unsigned int lefto(unsigned int __a);`

  Returns the number of leading ones in operand `__a`. This built-in function generates the `lefto` instruction.

Multiplier Built-In Functions

The SHARC compiler provides a set of built-in functions that can be used to directly generate instructions that target the multiplier registers `MRF` and `MRB`. These built-in functions use the `acc80` type, which can be used to store the value of the multiplier registers but cannot be used to perform any arithmetic using standard C operators such as `+`, `-` or `*`.

The following built-in functions may be used to initialize `acc80` values and are declared in `builtins.h`:

- `acc80 A_init(long long __a);`

  Returns an `acc80`-typed result containing initialized to the value `__a`, suitable for use in subsequent multiplier built-in functions.

- `acc80 A_zero(void);`

  Returns an `acc80`-typed result containing the value zero, suitable for use in subsequent multiplier built-in functions.

- `acc80 A_mr0(acc80 __a, unsigned int __b);`

  Sets the bottom 32 bits of `__a` to the value held in `__b`.

  This built-in function generates a copy into either the `MR0F` or `MR0B` register.

- `acc80 A_mr1(acc80 __a, unsigned int __b);`

  Sets bits 32 to 63 of `__a` to the value held in `__b`.

  This built-in function generates a copy into either the `MR1F` or `MR1B` register.

- `acc80 A_mr2(acc80 __a, int __b);`

  Sets bits 64 to 79 of __a to the value held in __b. This built-in function generates a copy into either the `MR2F` or `MR2B` register.

The following built-in functions may be used to multiply two values to produce an `acc80` result. They are declared in `builtins.h`:

- `acc80 A_mul_ssi(int __a, int __b);`

  Multiplies signed integers `__a` and `__b` to produce an `acc80`-typed result. This built-in function generates the `(SSI)` variant of the multiply instruction.

- `acc80 A_mul_sui(int __a, unsigned int __b);`

  Multiplies signed integer `__a` by unsigned integer `__b` to produce an `acc80`-typed result. This built-in function generates the `(SUI)` variant of the multiply instruction.

- `acc80 A_mul_usi(unsigned int __a, int __b);`

  Multiplies unsigned integer `__a` by signed integer `__b` to produce an `acc80`-typed result. This built-in function generates the `(USI)` variant of the multiply instruction.

- `acc80 A_mul_uui(unsigned int __a, unsigned int __b);`

  Multiplies unsigned integers `__a` and `__b` to produce an `acc80`-typed result. This built-in function generates the `(UUI)` variant of the multiply instruction.

- `acc80 A_mul_ssf(int __a, int __b);`

  Multiplies signed fractional values `__a` and `__b` to produce an `acc80`-typed result. This built-in function generates the `(SSF)` variant of the multiply instruction.

- `acc80 A_mul_suf(int __a, unsigned int __b);`

  Multiplies signed fractional value `__a` by unsigned fractional value `__b` to produce an `acc80`-typed result. This built-in function generates the `(SUF)` variant of the multiply instruction.

- `acc80 A_mul_usf(unsigned int __a, int __b);`

  Multiplies unsigned fractional value `__a` by signed fractional value `__b` to produce an `acc80`-typed result. This built-in function generates the `(USF)` variant of the multiply instruction.

- `acc80 A_mul_uuf(unsigned int __a, unsigned int __b);`

  Multiplies unsigned fractional values `__a` and `__b` to produce an `acc80`-typed result. This built-in function generates the `(UUF)` variant of the multiply instruction.

- `acc80 A_mul_ssfr(int __a, int __b);`

  Multiplies signed fractional values `__a` and `__b` to produce an `acc80`-typed result. This built-in function generates the `(SSFR)` variant of the multiply instruction.

- `acc80 A_mul_sufr(int __a, unsigned int __b);`

  Multiplies signed fractional value `__a` by unsigned fractional value `__b` to produce an `acc80`-typed result. This built-in function generates the `(SUFR)` variant of the multiply instruction.

- `acc80 A_mul_usfr(unsigned int __a, int __b);`

Multiplies unsigned fractional value __a by signed fractional value __b to produce an acc80-typed result. This built-in function generates the (USFR) variant of the multiply instruction.

- acc80 A_mul_uufr(unsigned int __a, unsigned int __b); Multiplies unsigned fractional values __a and __b to produce an acc80-typed result. This built-in function generates the (UUFR) variant of the multiply instruction.

The following built-in functions may be used to perform multiply-accumulates, multiplying two values and adding the result to an acc80 operand to produce an acc80 result. They are declared in builtins.h:

- acc80 A_mac_ssi(acc80 __a, int __b, int __c);

  Multiplies signed integers __b and __c and adds the product to __a to produce an acc80-typed result. This built-in function generates the (SSI) variant of the multiply-accumulate instruction.

- acc80 A_mac_sui(acc80 __a, int __b, unsigned int __c);

  Multiplies signed integer __b by unsigned integer __c and adds the product to __a to produce an acc80-typed result. This built-in function generates the (SUI) variant of the multiply-accumulate instruction.

- acc80 A_mac_usi(acc80 __a, unsigned int __b, int __c);

  Multiplies unsigned integer __b by signed integer __c and adds the product to __a to produce an acc80-typed result. This built-in function generates the (USI) variant of the multiply-accumulate instruction.

- acc80 A_mac_uui(acc80 __a, unsigned int __b, unsigned int __c);

  Multiplies unsigned integers __b and __c and adds the product to __a to produce an acc80-typed result. This built-in function generates the (UUI) variant of the multiply-accumulate instruction.

- acc80 A_mac_ssf(acc80 __a, int __b, int __c);

  Multiplies signed fractional values __b and __c and adds the product to __a to produce an acc80-typed result. This built-in function generates the (SSF) variant of the multiply-accumulate instruction.

- acc80 A_mac_suf(acc80 __a, int __b, unsigned int __c);

  Multiplies signed fractional value __b by unsigned fractional value __c and adds the product to __a to produce an acc80-typed result. This built-in function generates the (SUF) variant of the multiply-accumulate instruction.

- acc80 A_mac_usf(acc80 __a, unsigned int __b, int __c);

  Multiplies unsigned fractional value __b by signed fractional value __c and adds the product to __a to produce an acc80-typed result. This built-in function generates the (USF) variant of the multiply-accumulate instruction.

- acc80 A_mac_uuf(acc80 __a, unsigned int __b, unsigned int __c);

  Multiplies unsigned fractional values __b and __c and adds the product to __a to produce an acc80-typed result. This built-in function generates the (UUF) variant of the multiply-accumulate instruction.

- `acc80 A_mac_ssfr(acc80 __a, int __b, int __c);`

  Multiplies signed fractional values `__b` and `__c` and adds the product to `__a` to produce an `acc80`-typed result. This built-in function generates the `(SSFR)` variant of the multiply-accumulate instruction.

- `acc80 A_mac_sufr(acc80 __a, int __b, unsigned int __c);`

  Multiplies signed fractional value `__b` by unsigned fractional value `__c` and adds the product to `__a` to produce an `acc80`-typed result. This built-in function generates the `(SUFR)` variant of the multiply-accumulate instruction.

- `acc80 A_mac_usfr(acc80 __a, unsigned int __b, int __c);`

  Multiplies unsigned fractional value `__b` by signed fractional value `__c` and adds the product to `__a` to produce an `acc80`-typed result. This built-in function generates the `(USFR)` variant of the multiply-accumulate instruction.

- `acc80 A_mac_uufr(acc80 __a, unsigned int __b, unsigned int __c);`

  Multiplies unsigned fractional values `__b` and `__c` and adds the product to `__a` to produce an `acc80`-typed result. This built-in function generates the `(UUFR)` variant of the multiply-accumulate instruction.

The following built-in functions may be used to perform multiply-subtracts, multiplying two values and subtracting the result from an `acc80` operand to produce an `acc80` result. They are declared in `builtins.h`:

- `acc80 A_msub_ssi(acc80 __a, int __b, int __c);`

  Multiplies signed integers `__b` and `__c` and subtracts the product from `__a` to produce an `acc80`-typed result. This built-in function generates the `(SSI)` variant of the multiply-accumulate instruction.

- `acc80 A_msub_sui(acc80 __a, int __b, unsigned int __c);`

  Multiplies signed integer `__b` by unsigned integer `__c` and subtracts the product from `__a` to produce an `acc80`-typed result. This built-in function generates the `(SUI)` variant of the multiply-accumulate instruction.

- `acc80 A_msub_usi(acc80 __a, unsigned int __b, int __c);`

  Multiplies unsigned integer `__b` by signed integer `__c` and subtracts the product from `__a` to produce an `acc80`-typed result. This built-in function generates the `(USI)` variant of the multiply-accumulate instruction.

- `acc80 A_msub_uui(acc80 __a, unsigned int __b, unsigned int __c);`

  Multiplies unsigned integers `__b` and `__c` and subtracts the product from `__a` to produce an `acc80`-typed result. This built-in function generates the `(UUI)` variant of the multiply-accumulate instruction.

- `acc80 A_msub_ssf(acc80 __a, int __b, int __c);`

  Multiplies signed fractional values `__b` and `__c` and subtracts the product from `__a` to produce an `acc80`-typed result. This built-in function generates the `(SSF)` variant of the multiply-accumulate instruction.

- `acc80 A_msub_suf(acc80 __a, int __b, unsigned int __c);`

Multiplies signed fractional value __b by unsigned fractional value __c and subtracts the product from __a to produce an acc80-typed result. This built-in function generates the (SUF) variant of the multiply-accumulate instruction.

- `acc80 A_msub_usf(acc80 __a, unsigned int __b, int __c);`

  Multiplies unsigned fractional value __b by signed fractional value __c and subtracts the product from __a to produce an acc80-typed result. This built-in function generates the (USF) variant of the multiply-accumulate instruction.

- `acc80 A_msub_uuf(acc80 __a, unsigned int __b, unsigned int __c);`

  Multiplies unsigned fractional values __b and __c and subtracts the product from __a to produce an acc80-typed result. This built-in function generates the (UUF) variant of the multiply-accumulate instruction.

- `acc80 A_msub_ssfr(acc80 __a, int __b, int __c);`

  Multiplies signed fractional values __b and __c and subtracts the product from __a to produce an acc80-typed result. This built-in function generates the (SSFR) variant of the multiply-accumulate instruction.

- `acc80 A_msub_sufr(acc80 __a, int __b, unsigned int __c);`

  Multiplies signed fractional value __b by unsigned fractional value __c and subtracts the product from __a to produce an acc80-typed result. This built-in function generates the (SUFR) variant of the multiply-accumulate instruction.

- `acc80 A_msub_usfr(acc80 __a, unsigned int __b, int __c);`

  Multiplies unsigned fractional value __b by signed fractional value __c and subtracts the product from __a to produce an acc80-typed result. This built-in function generates the (USFR) variant of the multiply-accumulate instruction.

- `acc80 A_msub_uufr(acc80 __a, unsigned int __b, unsigned int __c);`

  Multiplies unsigned fractional values __b and __c and subtracts the product from __a to produce an acc80-typed result. This built-in function generates the (UUFR) variant of the multiply-accumulate instruction.

The following built-in functions may be used to saturate or round an acc80 operand to produce an acc80 result. They are declared in builtins.h:

- `acc80 A_sat_si(acc80 __a);`

  Saturates __a to a 32-bit signed integer and produces an acc80-typed result. This built-in function generates the (SI) variant of the SAT instruction.

- `acc80 A_sat_ui(acc80 __a);`

  Saturates __a to a 32-bit unsigned integer and produces an acc80-typed result. This built-in function generates the (UI) variant of the SAT instruction.

- `acc80 A_sat_sf(acc80 __a);`

  Saturates `__a` to a 64-bit signed fractional value and produces an `acc80`-typed result. This built-in function generates the `(SF)` variant of the `SAT` instruction.

- `acc80 A_sat_uf(acc80 __a);`

  Saturates `__a` to a 64-bit unsigned fractional value and produces an `acc80`-typed result. This built-in function generates the `(UF)` variant of the `SAT` instruction.

- `acc80 A_rnd_sf(acc80 __a);`

  Rounds `__a` to a 32-bit signed fractional value and produces an `acc80`-typed result. This built-in function generates the `(SF)` variant of the `RND` instruction.

- `acc80 A_rnd_uf(acc80 __a);`

  Rounds `__a` to a 32-bit unsigned fractional value and produces an `acc80`-typed result. This built-in function generates the `(UF)` variant of the `RND` instruction.

The following built-in functions may be used to extract a value from an `acc80` operand to produce an integer result, with optional saturation or rounding. They are declared in `builtins.h`:

- `int sat_si(acc80 __a);`

  Saturates `__a` to a 32-bit signed integer. This built-in function generates the `(SI)` variant of the `SAT` instruction.

- `unsigned int sat_ui(acc80 __a);`

  Saturates `__a` to a 32-bit unsigned integer. This built-in function generates the `(UI)` variant of the `SAT` instruction.

- `int sat_sf(acc80 __a);`

  Saturates `__a` to a 32-bit signed fractional value. This built-in function generates the `(SF)` variant of the `SAT` instruction.

- `unsigned int sat_uf(acc80 __a);`

  Saturates `__a` to a 32-bit unsigned fractional value. This built-in function generates the `(UF)` variant of the `SAT` instruction.

- `int rnd_sf(acc80 __a);`

  Rounds `__a` to a 32-bit signed fractional value. This built-in function generates the `(SF)` variant of the `RND` instruction.

- `unsigned int rnd_uf(acc80 __a);`

  Rounds `__a` to a 32-bit unsigned fractional value. This built-in function generates the `(UF)` variant of the `RND` instruction.

- `unsigned int mr0(acc80 __a);`

Returns the bottom 32 bits of __a. This built-in function generates a copy from either the `MR0F` or `MR0B` register.

- `unsigned int mr1(acc80 __a);`

  Returns bits 32 to 63 of __a. This built-in function generates a copy from either the `MR1F` or `MR1B` register.

- `int mr2(acc80 __a);`

  Returns bits 64 to 79 of __a. This built-in function generates a copy from either the `MR2F` or `MR2B` register.

Exclusive Transaction Built-In Functions

Processors that support byte-addressing (shown in Processor Features, the ***Processors Supporting Byte-Addressing*** table) also support instructions that perform exclusive transactions to memory. Built-in functions are provided to allow use of these accesses from C/C++ code. These are given in the ***Built-in Functions for Exclusive Memory Accesses*** table.

Table 2-38: Built-in Functions for Exclusive Memory Accesses

Built-in function	*Description*
`unsigned char load_exclusive_8(volatile unsigned char *, int *)`	Load and return 8-bit value from address given by first argument using exclusive access. Set memory pointed to by second parameter to zero on success, one on failure. Not available when using the `-char-size-32` compiler switch.
`unsigned short load_exclusive_16(volatile unsigned short *, int *)`	Load and return 16-bit value from address given by first argument using exclusive access. Set memory pointed to by second parameter to zero on success, one on failure. Not available when using the `-char-size-32` compiler switch.
`unsigned int load_exclusive_32(volatile unsigned int *, int *)`	Load and return 32-bit value from address given by first argument using exclusive access. Set memory pointed to by second parameter to zero on success, one on failure.
`unsigned long long load_exclusive_64(volatile unsigned long long *, int *)`	Load and return 64-bit value from address given by first argument using exclusive access. Set memory pointed to by second parameter to zero on success, one on failure.
`int store_exclusive_8(unsigned char, volatile unsigned char *)`	Store 8-bit value (first argument) to address (second argument) using exclusive access. Zero return value indicates success. Not available when using the `-char-size-32` compiler switch.
`int store_exclusive_16(unsigned short, volatile unsigned short *)`	Store 16-bit value (first argument) to address (second argument) using exclusive access. Zero return value indicates success. Not available when using the `-char-size-32` compiler switch.
`int store_exclusive_32(unsigned int, volatile unsigned int *)`	Store 32-bit value (first argument) to address (second argument) using exclusive access. Zero return value indicates success.
`int store_exclusive_64(unsigned long long, volatile unsigned long long *)`	Store 64-bit value (first argument) to address (second argument) using exclusive access. Zero return value indicates success.

The exclusive load built-in functions return the value held in the memory location indicated by their first parameter. A second parameter is used so that the load can return whether the address being accessed supports exclusive

accesses. A value of zero indicates success, while one indicates that exclusive accesses are not supported to the given address.

The exclusive store built-in functions attempt to store the value given by the first argument to the memory address given by the second argument. If the memory has not been written to since the preceding exclusive load of the same address, the value is written to the memory location and the function returns zero. Otherwise the memory is not updated and the function returns a non-zero result. An example of using these built-in functions to test and obtain a spin lock is given in the following example.

Example. Acquiring a spin lock using the exclusive transaction built-in functions.

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

void acquire_spin_lock(volatile unsigned int *lock) {
  while (1) {
    int err;
    int locked = load_exclusive_32(lock, &err);
    if (err != 0)
      abort();
    if (locked == 0) {
      err = store_exclusive_32(1, lock);
      if (err == 0)
        break;
    }
  }
}
```

Miscellaneous Built-In Functions

- ```
  unsigned int iop_read(volatile void * __a);
  ```

  The `iop_read` built-in function performs a volatile read from the address given by the argument, returning an `unsigned int`. It can be used when the `-no-assume-vols-are-iops` compiler switch is used. In this case, the built-in function can be used to access a memory-mapped IOP register (and have any necessary anomaly workarounds applied) while other volatile accesses are still treated by the compiler as non-IOP accesses.

- ```
  void iop_write(volatile void * __a, unsigned int  __b);
  ```

  The `iop_write` built-in function performs a volatile write of the second argument to the address given by the first argument. It can be used when the `-no-assume-vols-are-iops` compiler switch is used. In this case, the built-in function can be used to access a memory-mapped IOP register (and have any necessary anomaly workarounds applied) while other volatile accesses are still treated by the compiler as non-IOP accesses.

- ```
  void NOP(void);
  ```

  The `NOP` built-in function inserts a `NOP` instruction.
  ```

- `int funcsize(const void *func);`

  The `funcsize` built-in function returns the size of function in instruction words. The result is calculated from the difference between the start and end labels for the function operand. The compiler creates these labels for all C/C++ functions.

  The start label is the mangled name of the function. The end label used is a dot ("`.`") followed by the start label followed by "`.end`". For example, for C function `foo` compiled with `-char-size-32`, these labels are "`_foo:`" and "`._foo.end:`".

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

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

Pragmas

The 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 following sections describe the pragmas supported by the compiler:

- Data Declaration Pragmas
- Interrupt Handler Pragmas
- Loop Optimization Pragmas
- General Optimization Pragmas
- Function Side-Effect Pragmas
- Class Conversion Optimization Pragmas

- Template Instantiation Pragmas
- Header File Control Pragmas
- Fixed-Point Arithmetic Pragmas
- Inline Control Pragmas
- Linking Control Pragmas
- Diagnostic Control Pragmas
- Run-Time Checking Pragmas
- Code Generation Pragmas
- Exceptions Table Pragma
- Mixed Char-Size Interface Pragmas

The compiler issues a warning when it encounters an unrecognized pragma directive or pragma operator. Refer to the Optimal Performance from C/C++ Source Code chapter on how to use pragmas for code optimization.

Data Declaration Pragmas

The data declaration pragmas include `align`, `alignment_region`, `pack`, `pad`, and `no_partial_initialization` 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 *alignopt*

The `align`*`alignopt`* pragma can be used before variable and field declarations. It applies to the variable or field declaration that immediately follows the pragma.

The pragma's effect is that the next variable or field declaration should be forced to be aligned on a boundary specified by *`alignopt`*. The *`alignopt`* parameter is specified in addressable units, so when using the `-char-size-8` switch on processors that support byte-addressing, it should be specified in bytes, otherwise it should be specified in words.

- If the pragma is being applied to a local variable (since local variables are stored on the stack), the variable will only be changed when *`alignopt`* is not greater than the greatest possible stack alignment. If *`alignopt`* is greater than 2 words or 8 bytes, then a warning is given that the pragma is being ignored. Note that the variable is aligned relative to the stack pointer, and therefore for the pragma to be honored you must always make sure the stack pointer is kept double-word aligned. Therefore do not use the `-no-aligned-stack` switch and make sure interrupt handlers and assembly functions preserve double-word alignment of the stack pointer. For more details, see Stack Alignment and `-no-aligned-stack`.
- If *`alignopt`* 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 *`alignopt`*.

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

For example,

```
typedef struct {
   #pragma align 8
   int foo;
   int bar;

   #pragma align 8
   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

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

```
#pragma align _LONG
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 *alignopt* parameter is specified in addressable units, so when using the `-char-size-8` switch on processors that support byte-addressing, it should be specified in bytes, otherwise it should be specified in words. 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:

```
/* For processor without byte-addressing support or compiled
   with the -char-size-32 switch */
```

```
#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*)

The `pack` *alignopt* 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 *alignopt* parameter is specified in addressable units, so when using the `-char-size-8` switch on processors that support byte-addressing, it should be specified in bytes, otherwise it should be specified in words. 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*)

The `pad`*alignopt* pragma can be applied to `struct` definitions. It applies to all struct definitions that follow, until the default alignment is restored by omitting *alignopt*. The pragma is 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*. Specify the *alignopt* parameter

in addressable units. When using the -char-size-8 switch for processors supporting byte-addressing, specify *alignopt* in bytes; otherwise, specify *alignopt* in words.

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

NOTE: 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(8)
struct {
   int i;
   int j;
} s = {1,2};
#pragma pad()
```

#pragma no_partial_initialization

The no_partial_initialization pragma indicates that the compiler should raise a diagnostic if the following structure declaration does not provide an initialization value for all members of the structure. The pragma is useful when a structure declaration is extended between revisions of the software.

The following example shows how to use #pragma no_partial_initialization:

```
struct no_err {
  int x;
  int y;
};
#pragma no_partial_initialization

struct with_err {
  int x;
  int y;

};
struct no_err s1 = { 5 };   // no diagnostic
struct with_err s2 = { 5 }; // diagnostic reported
```

Interrupt Handler Pragmas

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

NOTE: It will not normally be necessary to use these pragmas when writing interrupt handlers; the standard interrupt dispatcher will be a more convenient approach. For details on writing and installing interrupt handlers, refer to the *System Run-Time Documentation* in the CCES online help.

NOTE: These pragmas are not supported for SHARC+ processors. Use of the pragmas on SHARC+ processors will result in a compile-time error.

CAUTION: When using these interrupt pragmas, you must use the correct pragmas for your application's context. Do not use these pragmas unless you know which registers require saving, are comfortable with modifying and rebuilding your startup code, and can ensure that your interrupt handlers will comply with the requirements of any RTOS you are using.

#pragma flush_restore_loop_stack

When the `flush_restore_loop_stack` pragma is applied to an interrupt handler, the compiler generates code to save the current loop status registers (`CURLCTR` and `LADDR`) to the stack, and to empty the loop stack, and to restore the loop stack at the end of the handler.

The pragma takes effect only when it is used in conjunction with one of `#pragma interrupt_complete` or `#pragma interrupt_complete_nesting`, otherwise an error message is issued.

#pragma implicit_push_sts_handler

When the `implicit_push_sts_handler` pragma is applied to an interrupt handler, the compiler does not generate an explicit `PUSH` and `POP` of `STS`.

The pragma takes effect only when it is used with `#pragma interrupt_complete` or `#pragma interrupt_complete_nesting`; otherwise, an error message is issued.

The compiler cannot determine whether the handler for the pragma is applied as a handler for the `VIRPT`, `IRQ`, or timer interrupts. It is your responsibility to determine whether to use the pragma or not.

#pragma interrupt_complete

The `interrupt_complete` pragma is similar to the `#pragma interrupt_complete_nesting` pragma, except that it does not re-enable interrupts. (It is for non-nested interrupt handlers.) This is done by not modifying the `MODE1` register.

By default, the pragma saves and restores only the top 32 bits of each data register. See #pragma save_restore_40_bits and #pragma save_restore_simd_40_bits for information on saving all 40 bits of the data registers.

NOTE: It will not normally be necessary to use this pragma when writing interrupt handlers; the standard interrupt dispatcher will be a more convenient approach. For details on writing and installing interrupt handlers, refer to the *System Run-Time Documentation* in the CCES online help.

NOTE: This pragma are not supported for SHARC+ processors. Use of the pragma on SHARC+ processors will result in a compile-time error.

CAUTION: Do not use this pragma unless you know which registers require saving, are comfortable with modifying and rebuilding your startup code, and can ensure that your interrupt handlers will comply with the requirements of any RTOS you are using.

#pragma interrupt_complete_nesting

The `interrupt_complete_nesting` pragma is used before a function definition. Use a function definition as an interrupt handler that can be called directly from the interrupt vector table. It produces a function that terminates with a "return from interrupt" sequence. Similar to `#pragma interrupt`, it saves and restores all registers

used by the function. It also performs a `PUSH STS` instruction at the start of the function to save the `MODE1` registers.

Since `PUSH STS` disables nested interrupts, and the pragma can be used with nested interrupts, it re-enables interrupts by the `BIT SET MODE1 0x1000;` instruction. At the end of the function, it performs a `POP STS` instruction to restore the status and `MODE1` registers.

By default, the pragma saves and restores only the top 32 bits of each data register. See #pragma save_restore_40_bits and #pragma save_restore_simd_40_bits for information on saving all 40 bits of the data registers.

NOTE: It will not normally be necessary to use this pragma when writing interrupt handlers; the standard interrupt dispatcher will be a more convenient approach. For details on writing and installing interrupt handlers, refer to the *System Run-Time Documentation* in the CCES online help.

NOTE: This pragma are not supported for SHARC+ processors. Use of the pragma on SHARC+ processors will result in a compile-time error.

CAUTION: Do not use this pragma unless you know which registers require saving, are comfortable with modifying and rebuilding your startup code, and can ensure that your interrupt handlers will comply with the requirements of any RTOS you are using.

#pragma interrupt_dispatched_handler

The `interrupt_dispatched_handler` pragma can be used before a function declaration or definition. It applies to the function declaration or definition that immediately follows the pragma.

The `interrupt_dispatched_handler` pragma is used for functions that will be called by an interrupt dispatcher. It 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 generated function uses the normal function return sequence; it does not use a "return from interrupt" sequence.

NOTE: It will not normally be necessary to use this pragma when writing interrupt handlers; the standard interrupt dispatcher will be a more convenient approach. For details on writing and installing interrupt handlers, refer to the *System Run-Time Documentation* in the CCES online help.

CAUTION: Do not use this pragma unless you know which registers require saving, are comfortable with modifying and rebuilding your startup code, and can ensure that your interrupt handlers will comply with the requirements of any RTOS you are using. The standard interrupt dispatcher provided by `adi_int.h` must be used if you use the uCOS2 or uCOS3 RTOS.

#pragma interrupt_reentrant

The `interrupt_reentrant` pragma is used in conjunction with `#pragma interrupt_dispatched_handler`, to generate a function that re-enables interrupts, so that the low-level dispatched handler can be interrupted by higher-priority interrupts.

#pragma save_restore_40_bits

The `save_restore_40_bits` pragma is used along with `#pragma interrupt_complete` or `#pragma interrupt_complete_nesting` to save and restore all 40 bits of the data registers (`Dregs`) used by the handler. This ensures that any routines using 40-bit arithmetic that are interrupted do not suffer accuracy problems. For leaf routines (that is, routines that do not call any other functions), the compiler saves and restores only the registers that are used. For non-leaf routines, the compiler saves and restores 40 bits of all `Dregs`. Note that saving and restoring each `Dregs` requires six instructions.

#pragma save_restore_simd_40_bits

The `save_restore_simd_40_bits` pragma is used with `#pragma interrupt_complete` and `#pragma interrupt_complete_nesting` to save and restore all 40 bits of any `PEx` data registers (`Dregs`) and `PEy` data registers (`Sregs`) used by the handler. It ensures that any routines using 40-bit arithmetic and SIMD mode that are interrupted do not suffer accuracy problems. For leaf routines (that is, routines that do not call any other functions), the compiler saves and restores only the registers that are used. For non-leaf routines, the compiler saves and restores 40 bits of all `Dregs` and `Sregs`. Saving and restoring each `Dregs` and `Sregs` requires six instructions.

NOTE: Only one run-time library routine (`cfft_mag()`) uses 40-bit arithmetic and SIMD mode.

Loop Optimization Pragmas

Loop optimization pragmas give the compiler additional information about the properties of the code 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. 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 to increase the cases where it knows that it is safe to do so. See Interprocedural Analysis for more information.

#pragma SIMD_for

The `SIMD_for` pragma must precede a `for`, `while`, or `do.. while` loop construct, and informs the compiler that the loop fulfills these conditions:

- Memory accesses are suitably aligned where necessary for SIMD mode.
- There are no memory accesses that rely on data stored during the previous iteration of the loop.
- There are no memory accesses within consecutive iterations that alias each other.
- Special constraints hold for any circular buffers used in the loop. The circular buffer length must consist of an even number of elements, and the initial value of the pointer must be aligned to point to an even element.

See SIMD Support for more details.

#pragma all_aligned

The `all_aligned` pragma applies to the subsequent loop. The pragma tells the compiler that all pointer-induction variables in the loop are initially 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

When the `no_vectorization` pragma is specified on a loop, it ensures the compiler does not generate vectorized SIMD code for the loop.

The pragma may also be specified on a function definition. For more information, see #pragma no_vectorization.

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

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

For example,

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

#pragma loop_unroll *N*

The `loop_unroll pragma` can be used only before a `for`, `while` or `do.. while` loop. The pragma takes exactly one positive integer argument, `N`, and it instructs the compiler to unroll the loop *N* times prior to further transforming the code.

In the most general case, the effect of:

```
#pragma loop_unroll N

for (init statements;  condition;  increment code) {
   loop_body
}
```

is equivalent to transforming the loop to:

```
for (init statements;  condition;  increment code) {
   loop_body   /* copy 1 */
   increment_code
      if (!condition)
         break;

   loop_body   /* copy 2 */
   increment_code
      if (!condition)
```

```
         break;
      ...

   loop_body   /* copy N-1 */
   increment_code
      if (!condition)
         break;
   loop_body   /* copy N */
```

Similarly, the effect of:

```
#pragma loop_unroll N
while (condition) {
   loop_body
}
```

is equivalent to transforming the loop to:

```
while (condition) {
   loop_body   /* copy 1 */
      if (!condition)
         break;

   loop_body   /* copy 2 */
      if (!condition)
         break;
      ...

   loop_body   /* copy N-1 */
      if (!condition)
         break;
   loop_body   /* copy N */
}
```

and the effect of:

```
#pragma loop_unroll N
do {
   loop_body
} while (condition)
```

is equivalent to transforming the loop to:

```
do {
   loop_body   /* copy 1 */
      if (!condition)
         break;

   loop_body   /* copy 2 */
      if (!condition)
         break;
      ...
```

```
   loop_body   /* copy N-1 */
      if (!condition)
         break;

   loop_body   /* copy N */
} while (condition)
```

#pragma no_alias

Use the `no_alias` pragma to inform the compiler that the following loop has no loads or stores that reference the same memory as each other. 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 can 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

The `vector_for` pragma tells the compiler that all iterations of the loop can run in parallel with each other, and data accessed in the loop are aligned suitably for SIMD operation. 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 it has the various properties necessary for the vectorization transformation.

The pragma has two effects:

- It asserts to the compiler that data accesses are suitably aligned for SIMD operation.
- It 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 one of the memory accesses were to have a non-unit stride through the array), 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`

  The pragma turns off the optimizer, if it was enabled, meaning it has the same effect as compiling with no optimization enabled.

- `#pragma optimize_for_space`

  The 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`

  The 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`

  The 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`, `compatible_pm_dm_params`, `pure`, `const`, `regs_clobbered`, `misra_func`, `no_vectorization`, `noreturn`, `exceptret`, `overlay`, `pgo_ignore` 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);
```

The `regs_clobbered_call` pragma is used before function call statement to give information regarding that particular call site.

#pragma alloc

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

```
#define N 100
#pragma allocint *new_buf(void);

int *vmul(int *a, int *b) {
   int i;   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 compatible_pm_dm_params

The `compatible_pm_dm_params` pragma tells the compiler to treat `pm`- and `dm`-qualified pointers as assignment-compatible within the following function declaration. The pragma is ignored if it does not immediately precede a function declaration.

See also `-compatible-pm-dm`.

#pragma const

The `const` 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 solely a function of its parameter values - two invocations with identical parameters will always return the same result. If any of the parameters are pointers, the function may not read the data they point at.

#pragma exceptret

The `exceptret` pragma can be placed before a function prototype or definition. The pragma tells the compiler that the function to which it applies will not return normally, but that it might throw a C++ exception.

The use of this pragma allows the compiler to treat any code, other than exception handlers, that follows a call to a function declared with the pragma as unreachable and hence removable.

```
#pragma exceptret
void fail() {
   throw std::runtime_error("Something went wrong.");
```

```
}

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

#pragma misra_func(arg)

The `misra_func(arg)` pragma is placed before a function prototype. It is used to support MISRA-C rules 20.4, 20.7, 20.8, 20.9, 20.10, 20.11, and 20.12. The `arg` indicates the type of function with respect to the MISRA-C Rule. Functions following Rule 20.4 would take arg `heap`, 20.7 arg `jmp`, 20.8 arg `handler`, 20.9 arg `io`, 20.10 arg `string_conv`, 20.11 arg `system` and 20.12 arg `time`.

#pragma no_vectorization

When the `no_vectorization` pragma is specified immediately before a function definition, it ensures the compiler does not generate vectorized SIMD code for any loop in the function on which it is specified.

The pragma may also be specified on individual loops.

#pragma noreturn

The `noreturn` pragma can be placed before a function prototype or definition. The pragma 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. If the function can throw a C++ exception, the `#pragma exceptret` pragma should be used instead.

The 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 overlay

When compiling code which involves one function calling another in the same source file, the compiler optimizer can propagate register information between the functions. This means that it can record which scratch registers are clobbered over the function call. This can cause problems when compiling overlaid functions, as the compiler may assume that certain scratch registers are not clobbered over the function call, but they are clobbered by the overlay manager. The `#pragma overlay`, when placed on the definition of a function, will disable this propagation of register information to the function's callers.

For example,

```
#pragma overlay
int add(int a, int b)
{
 // callers of function add() assume it clobbers
 // all scratch registers
 return a+b;
}
```

#pragma pgo_ignore

The `pgo_ignore` pragma tells the compiler that no profile should be generated for this function, when using profile-guided optimization. This is useful when the function is concerned with error checking or diagnostics.

```
extern const short *x, *y;
int dotprod(void) {
   int i, sum = 0;
   for (i = 0; i < 100; i++)
      sum += x[i] * y[i];
   return sum;
}

#pragma pgo_ignore
int check_dotprod(void) {
   /* The compiler will not profile this comparison */
   return dotprod() == 100;
}
```

#pragma pure

The 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 = 0; 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*

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

When used with an external function declaration, the 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 i12"
// Function "g" will only clobber r3, m4, r5, and i12
int g(int a) {
   return a+3;
}
```

NOTE: 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.

NOTE: Care must be taken when using the regs_clobbered pragma with pointers to functions. A function pointer cannot be declared to have a customized clobber set, so if you take the address of a function which has a customized clobber set then you must also add #pragma regs_clobbered_call to every call through that the function pointer. The compiler raises a warning if you take the address of a function with a regs_clobbered pragma.

String Syntax

A regs_clobbered string consists of a list of registers, register ranges, or register sets that are clobbered. 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. The ***Clobbered Register Sets*** table shows defined clobbered register sets.

Table 2-39: Clobbered Register Sets

Set	*Registers*
`CCset`	ASTATx, ASTATy
`SHADOWset`	All S regs, all Shadow MR regs, ASTATy
`MRset`	MRF, MRB; shadow MRF, shadow MRB
`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
`everything`	All registers, apart from those that are user-reserved or unclobberable

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

Unclobberable and Must Clobber Registers

There are certain caveats as to what registers may or must be placed in the clobbered set. See the ***Clobbered Register Sets*** table. 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.

`I6, I7, B6, B7, L6, L7`

Registers from these classes,

`D, I, B, 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

Registers in the USTAT class and user-reserved registers, which are indicated via the -reserve switch (-reserve *register[, register …]*), 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 registers are not saved and restored. Only the return registers used by the particular function return type are special. Return registers used by different return types are treated in the clobbered list in the conventional 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 regs_clobbered_call string

The regs_clobbered_call pragma may be applied to a statement to indicate that the call within the statement uses a modified volatile register set. The pragma is closely related to #pragma regs_clobbered, but avoids some of the restrictions that relate to that pragma.

These restrictions arise because the regs_clobbered pragma applies to a function's declaration-when the call is made, the clobber set is retrieved from the declaration automatically. This is not possible when the declaration is not available, because the function being called is not directly tied to a declaration of a specific function. This affects:

- Pointers to functions
- Class methods
- Pointers to class methods
- Virtual functions

In such cases, the `regs_clobbered_call` pragma can be used at the call site to inform the compiler directly of the volatile register set to be used during the call.

The pragma's syntax is as follows:

```
#pragma regs_clobbered_call clobber_string
   statement
```

where *`clobber_string`* follows the same format as for the `regs_clobbered` pragma and *`statement`* is the C statement containing the call expression.

There must be only a single call within the statement; otherwise, the statement is ambiguous. For example,

```
#pragma regs_clobbered "r0 r1 r2 i12"
int func(int arg) { /* some code */ }

int (*fnptr)(int) = func;

int caller(int value) {
   int r;

   #pragma regs_clobbered_call "r0 r1 r2 i12"
   r = (*fnptr)(value);
   return r;
}
```

NOTE: When you use the `regs_clobbered_call` pragma, you must ensure that the called function does indeed only modify the registers listed in the clobber set for the call - the compiler does not check this for you. It is valid for the callee to clobber less than is listed in the call's clobber set. It is also valid for the callee to modify registers outside of the call's clobber set, as long as the callee saves the values first and restores them before returning to the caller.

The following examples show this.

Example 1.

```
#pragma regs_clobbered "r0 r1 r2 i12"
void callee(void) { ... }

#pragma regs_clobbered_call "r0 r1 r2 i12"
callee();    // Okay - clobber sets match
```

Example 2.

```
#pragma regs_clobbered "r0 r2 i12"
void callee(void) { ... }

#pragma regs_clobbered_call "r0 r1 r2 i12"
callee();    // Okay - callee clobber set is a subset
             // of call's set
```

Example 3.

```
#pragma regs_clobbered "r0 r1 r2 i12"
void callee(void) { ... }

#pragma regs_clobbered_call "r0 r2 i12"
callee();    // Error - callee clobbers more than
             // indicated by call.
```

Example 4.

```
void callee(void) { ... }

#pragma regs_clobbered_call "r0 r1 r2 i12"
callee();    // Error - callee uses default set larger
             // than indicated by call.
```

Limitations

Pragma `regs_clobbered_call` may not be used on constructors or destructors of C++ classes.

The pragma only applies to the call in the immediately-following statement. If the immediately-following line contains more than one statement, the pragma only applies to the first statement on the line:

```
#pragma regs_clobbered "r0 r1 r2 i12"
x = foo(); y = bar(); // only "x = foo();" is affected by
                      // the pragma.
```

Similarly, if the immediately-following line is a sequence of declarations that use calls to initialize the variables, then only the first declaration is affected:

```
#pragma regs_clobbered "r0 r1 r2 i12"
int x = foo(), y = bar();// only "x = foo()" is affected
                         // by the pragma.
```

Moreover, if the declaration with the call-based initializer is not the first in the declaration list, the pragma will have no effect:

```
#pragma regs_clobbered "r0 r1 r2 i12"
int w = 4, x = foo(); y = bar();// pragma has no effect
                                // on "w = 4".
```

The pragma has no effect on function calls that get inlined. Once a function call is inlined, the inlined code obeys the clobber set of the function into which it has been inlined. It does not continue to obey the clobber set that will be used if an out-of-line copy is required.

#pragma result_alignment (*n*)

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

The pragma is often used in conjunction with the `#pragma alloc` of custom-allocation functions that return pointers that are more strictly aligned than could be deduced from their type. The following example shows a use of the pragma. Note that the pragma will not change the alignment of data returned by the declared function. It is a guideline to the compiler.

```
#pragma result_alignment(8)
int * alloc_align8_data(unsigned long size);
```

Class Conversion Optimization Pragmas

The class conversion optimization pragmas (`param_never_null`, `suppress_null_check`) allow the compiler to generate more efficient code when converting class pointers from a pointer-to-derived-class to a pointer-to-base-class, by asserting that the pointer to be converted will never be a null pointer. This allows the compiler to omit the null check during conversion.

#pragma param_never_null *param_name [...]*

This pragma must immediately precede a function definition. It specifies a name or a list of space-separated names, which must correspond to the parameter names declared in the function definition. It checks that the named parameter is a class pointer type. Using this information it will generate more efficient code for a conversion from a pointer to a derived class to a pointer to a base class. It removes the need to check for the null pointer during the conversion.

For example,

```
#include <iostream>
using namespace std;
class A {
    int a;
};
class B {
    int b;
};
class C: public A, public B {
    int c;
};

C obj;
B *bpart = &obj;
bool fail = false;

#pragma param_never_null pc
void func(C *pc)
{
    B *pb;
    pb = pc;    /* without pragma the code generated has to
                   check for NULL */
    if (pb != bpart)
        fail = true;
}

int main(void)
{
    func(&obj);
    if (fail)
        cout << "Test failed" << endl;
    else
```

```
        cout << "Test passed" << endl;
    return 0;
}
```

#pragma suppress_null_check

This pragma must immediately precede an assignment of two pointers or a declaration list. If the pragma precedes an assignment it indicates that the second operand pointer is not null and generates more efficient code for a conversion from a pointer to a derived class to a pointer to a base class. It removes the need to check for the null pointer before assignment. On a declaration list it marks all variables as not being the null pointer. If the declaration contains an initialization expression, that expression is not checked for null.

For example,

```
#include <iostream>
using namespace std;
class A {
    int a;
};
class B {
    int b;
};
class C: public A, public B {
    int c;
};

C obj;
B *bpart = &obj;
bool fail = false;

void func(C *pc)
{
    B *pb;
    #pragma suppress_null_check
    pb = pc;    /* without pragma the code generated has to
                   check for NULL */
    if (pb != bpart)
        fail = true;
}

void func2(C *pc)
{
    #pragma suppress_null_check
    B *pb = pc, *pb2 = pc;  /* pragma means these initializations
                  need not check for NULL. It also marks pb and pb2
                  as never being NULL, so the compiler will not
                  generate NULL checks in class conversions using
                  these pointers. */
    if (pb != bpart || pb2 != bpart)
        fail = true;
}
```

```
int main(void)
{
    func(&obj);
    func2(&obj);
    if (fail)
       cout << "Test failed" << endl;
    else
       cout << "Test passed" << endl;
    return 0;
}
```

Template Instantiation Pragmas

The template instantiation pragmas (`instantiate`, `do_not_instantiate` and `can_instantiate`) 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 for more information on how the compiler handles templates.

These pragmas take the name of an instance as a parameter, as shown in the ***Instance Names*** table.

Table 2-40: 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)`

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

#pragma instantiate *instance*

The `instantiate` 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*

The 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*

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

NOTE: Currently, the can_instantiate 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 (no_implicit_inclusion, once, and system_header) help the compiler to handle header files.

#pragma no_implicit_inclusion

When the -c++ switch 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
```

which contains

```
#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 once

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

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

NOTE: 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

The `system_header` pragma identifies an `include` file as a file supplied with CCES. The compiler makes use of this information to help optimize uses of the supplied library functions and inline functions that these files define. The pragma should not be used in user application source.

Fixed-Point Arithmetic Pragmas

The compiler supports several pragmas which can change the semantics of arithmetic on the native fixed-point type, `fract`. These are `#pragma FX_CONTRACT {ON|OFF}` and `#pragma FX_ROUNDING_MODE {TRUNCATION|BIASED|UNBIASED}`. In addition, `#pragma STDC FX_FULL_PRECISION {ON|OFF|DEFAULT}` and `#pragma STDC FX_FRACT_OVERFLOW {SAT|DEFAULT}` are accepted by the compiler but have no effect on generated code.

These pragmas may be used at file scope, in which case they apply to all following functions until another pragma is respecified to change the pragma state. Alternatively, they may be specified in a `{ }` delimited scope (or compound statement), where they will temporarily override the current setting of the pragma's state until the end of the scope.

For more information, see Using Native Fixed-Point Types.

#pragma FX_CONTRACT {ON|OFF}

The `FX_CONTRACT {ON|OFF}` pragma may be used to control the precision of intermediate results of calculations on the native fixed-point type `fract`. If `FX_CONTRACT` is `ON`, where an intermediate result is not stored back to a named variable, the compiler may choose to keep the intermediate result in greater precision than that mandated by the ISO/IEC C Technical Report 18037. It will do this where maintaining the higher precision allows more efficient code to be generated.

When `FX_CONTRACT` is `OFF`, the compiler will adhere strictly to the ISO/IEC Technical Report 18037 and will convert all intermediate results to the type dictated in this standard before use.

The following example shows the use of the pragma.

```
fract mulsu(fract f1, unsigned fract f2) {
#pragma FX_CONTRACT ON
```

```
   return f1 * f2;   /* compiler creates signed-unsigned multiply */
}
```

The default state of the FX_CONTRACT pragma is ON.

#pragma FX_ROUNDING_MODE {TRUNCATION|BIASED|UNBIASED}

The FX_ROUNDING_MODE {TRUNCATION|BIASED|UNBIASED} pragma may be used to control the rounding mode used during calculations on the native fixed-point type fract.

When FX_ROUNDING_MODE is set to TRUNCATION, the exact mathematical result of a computation is rounded by truncating the least significant bits beyond the precision of the result type. This is equivalent to rounding towards negative infinity.

When FX_ROUNDING_MODE is set to BIASED, the exact mathematical result of a computation is rounded to the nearest value that fits in the result type. If the exact result lies exactly half-way between two consecutive values in the result type, the result is rounded up to the higher one.

When FX_ROUNDING_MODE is set to UNBIASED, the exact mathematical result of a computation is rounded to the nearest value that fits in the result type. If the exact result lies exactly half-way between two consecutive values in the result type, the result is rounded to the even value.

The following example shows the use of the pragma.

```
fract divide_biased(fract f1, fract f2) {
   #pragma FX_ROUNDING_MODE BIASED
   return f1 / f2;   /* compiler creates divide with biased rounding */
}
```

The default state of the FX_ROUNDING_MODE pragma is TRUNCATION.

#pragma STDC FX_FULL_PRECISION {ON|OFF|DEFAULT}

The STDC FX_FULL_PRECISION {ON|OFF|DEFAULT} pragma is used by the ISO/IEC Technical Report 18037 to permit an implementation to generate faster code for fixed-point arithmetic, but produce lower-accuracy results.

The compiler always produces full-accuracy results. Therefore, although the pragma is accepted by the compiler, the code generated will be the same regardless of the state of FX_FULL_PRECISION.

#pragma STDC FX_FRACT_OVERFLOW {SAT|DEFAULT}

The STDC FX_FRACT_OVERFLOW {SAT|DEFAULT} pragma is used by the ISO/IEC Technical Report 18037 to permit an implementation to generate code that does not saturate fract-typed results on overflow.

The fract arithmetic with the compiler always saturates on overflow. Therefore, although the pragma is accepted by the compiler, the code generated will be the same regardless of the state of FX_FRACT_OVERFLOW.

Inline Control Pragmas

The compiler supports two pragmas to control the inlining of code. These pragmas are #pragma always_inline, #pragma inline, and #pragma never_inline.

#pragma always_inline

This `always_inline` pragma may be applied to a function definition to indicate to the compiler that the function should always be inlined, and never called "out of line". The pragma may only be applied to function definitions with the `inline` qualifier, and may not be used on functions with variable-length argument lists. It is invalid for function definitions that have interrupt-related pragmas associated with them.

If the function in question has its address taken, the compiler cannot guarantee that all calls are inlined, so a warning is issued.

See Function Inlining for details of pragma precedence during inlining.

The following are examples of the `always_inline` pragma.

```
int func1(int a) {        // only consider inlining
   return a + 1;          // if -Oa switch is on
}

inline int func2(int b) {// probably inlined, if optimizing
   return b + 2;
}

#pragma always_inline
inline int func3(int c) {// always inline, even unoptimized
   return c + 3;
}

#pragma always_inline
int func4(int d) {// error: not an inline function
   return d + 4;
}
```

#pragma inline

The pragma instructs the compiler to inline the function if it is considered desirable. The pragma is equivalent to specifying the `inline` keyword, but may be applied when the `inline` keyword is not allowed (such as when compiling in MISRA-C mode). For more information, see MISRA-C Compiler.

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

#pragma never_inline

This pragma may be applied to a function definition to indicate to the compiler that function should always be called "out of line", and that the function's body should never be inlined.

The pragma may not be used on function definitions that have the `inline` qualifier. See Function Inlining for details of pragma precedence during inlining.

These are code examples for the `never_inline` pragma.

```
#pragma never_inline
int func5(int e) {// never inlined, even with -Oa switch
   return e + 5;
}

#pragma never_inline
inline int func5(int f) {// error: inline function
   return f + 6;
}
```

Linking Control Pragmas

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

#pragma linkage_name *identifier*

The 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 ();
```

See also #pragma function_name *identifier*.

#pragma function_name *identifier*

The pragma associates the identifier with the next external function declaration. The compiler uses the given identifier instead of the one used in the function declaration for any external references. However, unlike `#pragma linkage_name`, the compiler follows its usual conventions to mangle the new name, for example by prefixing a leading underscore. For example,

```
_Pragma("function_name realfuncname")
void funcname ();
```

See also #pragma linkage_name *identifier*.

#pragma core

When building a project that targets multiple processors or multiple cores on a processor, it is possible to create a link stage that produces executables for more than one core or processor. The interprocedural analysis (IPA) framework requires that some conventions be adhered to in order to successfully perform its analyses for such projects.

Because the IPA framework collects information about the whole program, including information on references which may be to definitions outside the current translation unit, the IPA framework must be able to distinguish these definitions and their references without ambiguity.

If any confusion were allowed about which definition a reference refers to, then the IPA framework could potentially cause bad code to be generated, or could cause translation units in the project to be continually recompiled ad infinitum. It is the global symbols that are really relevant in this respect. The IPA framework will correctly handle locals and static symbols because multiple definitions are not possible within the same file, so there can be no ambiguity.

In order to disambiguate all references and the definitions to which they refer, it is necessary to have a unique name for each definition within a given project. It is illegal to define two different functions or variables with the same name. This is illegal in single-core projects because this would lead to multiple definitions of a symbol and the link would fail. In multi-core projects, however, it may be possible to link a project with multiple definitions because one definition could be linked into each link project, resulting in a valid link. Without detailed knowledge of what actions the linker had performed, however, the IPA framework would not be able to disambiguate such multiple definitions. For this reason, to use the IPA framework, it is up to you to ensure unique names even in projects targeting multiple cores or processors.

There are a few cases for which it is not possible to ensure unique names in multi-core or multi-processor projects. One such case is `main`. Each processor or core will have its own `main` function, and these need to be disambiguated for the IPA framework to be able to function correctly. Another case is where a library (or the C run-time start-up) references a symbol which the user may wish to define differently for each core.

For this reason, CCES supports the `#pragma core(corename)`.

The pragma can be provided immediately prior to a definition or a declaration. The pragma allows you to give a unique identifier to each definition. It also allows you to indicate to which definition each reference refers. The IPA framework will use this core identifier to distinguish all instances of symbols with the same name and will therefore be able to carry out its analyses correctly.

NOTE: The `corename` specified should only consist of alphanumeric characters. The core name is case sensitive.

The pragma should be used:

- On every definition (not in a library) for which there needs to be a distinct definition for each core.
- On every declaration of a symbol (not in a library) for which the relevant definition includes the use of `#pragma core`. The core specified for a declaration must agree with the core specified for the definition.

It should be noted that the IPA framework will not need to be informed of any distinction if there are two identical copies of the same function or data with the same name. Functions or data that come from objects and that are duplicated in memory local to each core, for example, will not need to be distinguished. The IPA framework does not need to know exactly which instance each reference will get linked to because the information processed by the framework is identical for each copy. Essentially, the pragma only needs to be specified on items where there will be different functions or data with the same name incorporated into the executable for each core.

Here is an example of `#pragma core` usage to distinguish two different main functions:

```
/* foo.c */
#pragma core("coreA")
int main(void) {
    /* Code to be executed by core A */
}
/* bar.c */
#pragma core("coreB")
int main(void) {
    /* Code to be executed by core B */
}
```

Omitting either instance of the pragma will cause the IPA framework to issue a fatal error indicating that the pragma has been omitted on at least one definition.

Here is an example that will cause an error to be issued because the name contains a non-alphanumeric character:

```
#pragma core("core/A")
int main(void) {
    /* Code to executed on core A */
}
```

Here is an example where the pragma needs to be specified on a declaration as well as the definitions. There is a library which contains a reference to a symbol which is expected to be defined for each core. Two more modules define the `main` functions for the two cores. Two further modules, each only used by one of the cores, makes a reference to this symbol, and therefore requires use of the pragma. For example,

```
/* libc.c */
#include <stdio.h>
extern int core_number;
void print_core_number(void) {
    printf("Core %d\n", core_number);
}
/* maina.c */
extern void fooa(void)
#pragma core("coreA")
int core_number = 1;
#pragma core("coreA")
int main(void) {
    /* Code to be executed by core A */
    print_core_number();
    fooa();
}
/* mainb.c */
extern void foob(void)
#pragma core("coreB")
int core_number = 2;
#pragma core("coreB")
int main(void) {
    /* Code to be executed by core B */
    print_core_number();
    foob();
}
/* fooa.c */
#include <stdio.h>
#pragma core("coreA")
extern int core_number;
void fooa(void) {
    printf("Core: is core%c\n", `A' - 1 + core_number);
}
/* foob.c */
#include <stdio.h>
#pragma core("coreB")
```

```
extern int core_number;
void fooa(void) {
    printf("Core: is core%c\n", `A' - 1 + core_number);
}
```

In general, it will only be necessary to use `#pragma core` in this manner when there is a reference from outside the application (in a library, for example) where there is expected to be a distinct definition provided for each core, and where there are other modules that also require access to their respective definition. Notice also that the declaration of `core_number` in `lib.c` does not require use of the pragma because it is part of a translation unit to be included in a library.

A project that includes more than one definition of `main` will undergo some extra checking to catch problems that would otherwise occur in the IPA framework. For any non-template symbol that has more than one definition, the tool chain will fault any definitions that are outside libraries that do not specify a core name with the pragma. This check does not affect the normal behavior of the prelinker with respect to templates and in particular the resolution of multiple template instantiations.

To clarify:

Inside a library, `#pragma core` is not required on declarations or definitions of symbols that are defined more than once. However, a library can be responsible for forcing the application to define a symbol more than once (that is, once for each core). In this case, the definitions and declarations require the pragma to be used outside the library to distinguish the multiple instances.

It should be noted that the tool chain cannot check that uses of `#pragma core` are consistent. If you use the pragma inconsistently or ambiguously then the IPA framework may end up causing incorrect code to be generated or causing continual recompilation of the application's files.

It is also important to note that the pragma does not change the linkage name of the symbol it is applied to in any way.

For more information on IPA, see Interprocedural Analysis.

#pragma retain_name

The `retain_name` pragma indicates that the function or variable declaration that follows the pragma is not removed even though it apparently has no uses. Normally, when Interprocedural Analysis or linker elimination are enabled, the CCES tools will identify unused functions and variables, and will eliminate them from the resulting executable to reduce memory requirements. The `retain_name` pragma instructs the tools to retain the specified symbol, regardless.

The following example shows how to use the pragma.

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

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

```
}

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

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

The pragma is only valid for global symbols. It is not valid for the following kinds of symbols:

- Symbols with `static` storage class
- Function parameters
- Symbols with `auto` storage class (locals). These are allocated on the stack at runtime.
- Members/fields within a `struct/union/class`
- Type declarations

For more information on IPA, see Interprocedural Analysis.

#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 the categories listed under *`SECTKIND`*. Except for the `STI` category, the pragma remains in force for a section category until its next use with that particular category, or the end of the file. The `STI` is an exception, in that only one `STIdefault_section` can be specified and its scope is the entire file scope, not just the part following the use of `STI`. A warning is issued if several `STI` sections are specified in the same file.

The omission of a section name results in the default section being reset to be the section that was in use at the start of the file, which can be either a compiler default value, or a value set by the user through the `-section` command line switch (for example, `-section SECTKIND=SECTSTRING)`.

In all cases (including STI), the `default_section` pragma overwrites the value specified with the `-section` command-line switch.

```
#pragma default_section(DATA, "NEW_DATA1")
int x;
#pragma default_section(DATA, "NEW_DATA2")
int y=5;
#pragma default_section(DATA, "NEW_DATA3")
int z;
```

In this case y is placed in NEW_DATA2, because the definition of y is within its scope.

A default_section pragma can only be used at global scope, where global variables are allowed.

SECTKIND can be one of the following keywords found in the ***SECTKIND Keywords*** table.

Table 2-41: SECTKIND Keywords

Keyword	*Description*
CODE	Section is used to contain procedures and functions
ALLDATA	Shorthand notation for DATA, CONSTDATA, BSS, STRINGS and AUTOINIT
DATA	Section is used to contain "normal data"
CONSTDATA	Section is used to contain read-only data
BSZ	Section is used to contain zero-filled data
SWITCH	Section is used to contain jump-tables to implement C/C++ switch statements
VTABLE	Section is used to contain C++ virtual-function tables
STI	Section that contains code required to be executed by C++ initializations. For more information, see Constructors and Destructors of Global Class Instances.
STRINGS	Section that stores string literals
AUTOINIT	Contains data used to initialize aggregate autos.
PM_DATA	Section is used to contain normal data declared with pm keyword
PM_CONSTDATA	Section is used to contain read-only data declared with pm keyword

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

Changing one section kind has no effect on other section kinds. For instance, even though STRINGS and CONSTDATA are by default placed by the compiler in the same section, if CONSTDATA default_section is changed, the change has no effect on the STRINGS data.

Please note that ALLDATA is not a real section, but a rather pseudo-kind that stands for DATA, CONSTDATA, STRINGS, AUTOINIT and BSZ, and changing ALLDATA is equivalent to changing all of these section kinds. Therefore,

```
#pragma default_section(ALLDATA, params)
```

is equivalent to the sequence:

```
#pragma default_section(DATA, params)
#pragma default_section(CONSTDATA, params)
#pragma default_section(STRINGS, params)
#pragma default_section(AUTOINIT, params)
#pragma default_section(BSZ, params)
```

QUALIFIER can be one of the following keywords found in the ***QUALIFIER Keywords*** table.

Table 2-42: QUALIFIER Keywords

Keyword	*Description*
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
SW	Code is short-word (processors that support VISA execution only).
NW	Code is normal-word (processors that support VISA execution only).
DMAONLY	Section is located in memory that can only be accessed by DMA. On ADSP-2126x and certain ADSP-2136x processors, this keyword applies to external memory.

There may be any number of comma-separated section qualifiers within such pragmas, but they must not conflict with one another. Qualifiers must also be consistent across pragmas for identical section names, and omission of qualifiers is not allowed even if at least one such qualifier has appeared in a previous pragma for the same section. If any qualifiers have not been specified for a particular section by the end of the translation unit, the compiler uses default qualifiers appropriate for the target processor. The following specifies that f() should be placed in a section "foo", which is DOUBLEANY qualified:

```
#pragma section("foo", DOUBLEANY)
void f() {}
```

The compiler always tries to honor the section pragma as its highest priority, and the default_section pragma is always the lowest priority.

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 keyword and therefore is placed in the PM data section:

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

If the PM data is explicitly set as in this example,

```
#pragma default_section(PM_DATA, "pm_onion")
#pragma default_section(DATA, "onion")
int pm a = 4;
```

then the variable a gets placed in the pm_onion section.

The following code results in code in section "foo" being compiled as short-word code (processors that support VISA execution only):

```
#pragma section("foo", SW)
```

The following results in code in section "foo2" being compiled as normal word code (on processors that support VISA execution only):

```
#pragma default_section(CODE, "foo2", NW)
```

NOTE: In cases where a C++ STL object must be placed in a specific memory section, using #pragma section/default_section will not work. Instead, a non-default heap must be used, as explained in Allocating C++ STL Objects to a Non-Default Heap.

#pragma file_attr("name[=value]"[, "name[=value]" [...]])

This pragma directs the compiler to emit the specified attributes when it compiles a file containing the pragma. Multiple #pragma file_attr directives are allowed in one file.

If "=value" is omitted, the default value of "1" will be used.

NOTE: The value of an attribute is all the characters after the '=' symbol and before the closing '”' symbol, including spaces. A warning will be emitted by the compiler if you have a preceding or trailing space as an attribute value, as this is likely to be a mistake.

See File Attributes for more information on using attributes.

#pragma weak_entry

The weak_entry pragma may be used before a static variable or function declaration or definition. It applies to the function/variable declaration or definition that immediately follows the pragma. Use of the 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(){}
```

NOTE: When a symbol definition is weak, it may be discarded by the linker in favor of another definition of the same symbol. Therefore, if any modules in the application make use of the weak_entry pragma, interprocedural analysis is disabled because it would be unsafe for the compiler to predict which definition will be selected by the linker. For more information, see Interprocedural Analysis.

Diagnostic Control Pragmas

The compiler supports `#pragma diag` which allows selective modification of the severity of compiler diagnostic messages.

The directive has three forms:

- Modify the severity of specific diagnostics
- Modify the behavior of an entire class of diagnostics
- Save or restore the current behavior of all diagnostics

Modifying the Severity of Specific Diagnostics

This form of the directive has the following syntax:

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

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

Table 2-43: Keywords for Action Qualifier

Keyword	*Action*
suppress	Suppresses all instances of the diagnostic
remark	Changes the severity of the diagnostic to a remark.
annotation	Changes the severity of the diagnostic to an annotation.
warning	Changes the severity of the diagnostic to a warning.
error	Changes the severity of the diagnostic to an error.
restore	Restores the severity of the diagnostic to what it was originally at the start of compilation after all command-line options were processed.

If not in MISRA-C mode, the *diag* qualifier can be one or more comma-separated compiler diagnostic numbers without the preceding "cc" or 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, some diagnostics are *global* (for example, diagnostics emitted by the compiler back-end after lexical analysis and parsing, or before parsing begins), and these global diagnostics cannot have their severity overridden by the diagnostic control pragmas. To modify the severity of global diagnostics, use the diagnostic control switches. For more information, see -W{annotation|error|remark|suppress|warn} *number[, number ...]*.

In MISRA-C mode, the *diag* qualifier is a list of MISRA-C rule numbers in the form *misra_rule_number_6_3* and *misra_rule_number_19_4* for rules 6.3 and 19.4, and so on. Special cases are rules 10.1 and 10.2, which are both split into four distinct rule checks. For example, 10.1(c) should be stated as *misra_rule_10_1_c*. diag may also be the special token misra_rules_all, which specifies that the pragma applies to all MISRA-C rules.

The third optional argument is a string-literal to insert a comment regarding the use of the `#pragma diag`.

Modifying the Behavior of an Entire Class of Diagnostics

This form of the directive has the following syntax, which is not allowed when in MISRA-C mode:

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

The effects are as follows:

- `#pragma diag(errors)`

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

- `#pragma diag(remarks)`

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

- `#pragma diag(annotations)`

  The pragma can be used to enable all subsequent annotations and warnings (equivalent to the `-Wannotations` switch option).

- `#pragma diag(warnings)`

  The pragma can be used to restore the default behavior when the `-w`, `-Wremarks` and `-Wannotations` switches are not specified, which is to display warnings but inhibit remarks and annotations.

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)`

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

- `#pragma diag (pop)`

  The 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, unless in MISRA-C mode. Note that the error threshold (set by the `remarks`, `annotations`, `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: 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`").

Run-Time Checking Pragmas

Run-time checking pragmas allow you to control the compiler's generation of additional checking code. This code can test at runtime for common programming errors. The `-rtcheck` command-line switch and its related switches control which common errors are tested for. Use the command-line switches to enable run-time checking; once run-time checking is enabled, the run-time checking pragmas can be used to disable and re-enable checking, for specific functions.

This section describes the following pragmas:

- #pragma rtcheck(off)
- #pragma rtcheck(on)

NOTE: Run-time checking causes the compiler to generate additional code to perform the checks. This code has space and performance overheads. Use of run-time checking should be restricted to application development, and should not be used on applications for release.

#pragma rtcheck(off)

The `rtcheck(off)` pragma disables any run-time check code generation that has been enabled via command-line switches, such as `-rtcheck`. The pragma is only valid at file scope, and affects code generation for function definitions that follow.

The pragma has no effect on checks for stack overflow, or checks of heap operations. This is because such checks are provided by selecting alternative library support at link-time, and so apply to the whole application.

#pragma rtcheck(on)

The `rtcheck(on)` pragma re-enables any run-time check code generation that was enabled via command-line switches, such as `-rtcheck`. The pragma is only valid at file scope, and affects code generation for function definitions that follow. If no run-time checking was enabled by command-line switches, the pragma has no effect.

Memory Bank Pragmas

The memory bank pragmas provide additional performance characteristics for the memory areas used to hold code and data for the function.

By default, the compiler assumes that there are no external costs associated with memory accesses. This strategy allows optimal performance when the code and data are placed into high-performance internal memory. In cases where the performance characteristics of memory are known in advance, the compiler can exploit this knowledge to improve the scheduling of generated code.

#pragma code_bank(*bankname*)

The code_bank pragma informs the compiler that the instructions for the immediately-following function are placed in a memory bank called *bankname*. Without the pragma, the compiler assumes that instructions are placed into the default bank, if one has been specified; see Memory Bank Selection for details. When optimizing the function, the compiler is aware of attributes of memory bank *bankname*, and determines how long it takes to fetch each instruction from the memory bank.

If *bankname* is omitted, the instructions for the function are not considered to be placed into any particular bank.

In the following example, the add_slowly() function is placed into the "slowmem" bank, which may have different performance characteristics from the default code bank, into which add_quickly() is placed.

```
#pragma code_bank(slowmem)
int add_slowly (int x, int y) { return x + y; }
int add_quickly(int a, int b) { return a + b; }
```

#pragma data_bank(*bankname*)

The data_bank pragma informs the compiler that the immediately-following function uses the memory bank *bankname* as the model for memory accesses for non-local data that does not otherwise specify a memory bank; see Memory Bank Selection for details. Without the pragma, the compiler assumes that non-local data should use the default bank, if any has been specified, for behavioral characteristics.

If *bankname* is omitted, the non-local data for the function is not considered to be placed into any specific bank.

In both green_func() and blue_func() of the following example, i is associated with the memory bank "blue", and the retrieval and update of i are optimized to use the performance characteristics associated with memory bank "blue".

```
#pragma data_bank(green)
int green_func(void)
{
      extern int arr1[32];
      extern int bank("blue") i;
      i &= 31;
      return arr1[i++];
}
int blue_func(void)
{
      extern int arr2[32];
      extern int bank("blue") i;
      i &= 31;
   return arr2[i++];
}
```

The array arr1 does not have an explicit memory bank in its declaration. Therefore, it is associated with the memory bank "green", because green_func() has a specific default data bank. In contrast, arr2 is associated with the default data memory bank (if any), because blue_func() does not have a #pragma data_bank preceding it.

#pragma stack_bank(*bankname*)

The stack_bank pragma informs the compiler that all locals for the immediately-following function are to be associated with memory bank *bankname*, unless they explicitly identify a different memory bank. Without the pragma, all locals are assumed to be associated with the default stack memory bank, if any; see Memory Bank Selection for details.

If *bankname* is omitted, locals for the function are not considered to be placed into any particular bank.

In the following example, the dotprod() function places the sum and i values into memory bank "mystack", while fib() places r, a, and b into the default stack memory bank (if any), because there is no stack_bank pragma. The count_ticks() function does not declare any local data, but any compiler-generated local storage uses the "sysstack" memory bank's performance characteristics.

```
#pragma stack_bank(mystack)
short dotprod(int n, const short *x, const short *y)
{
      int sum = 0;
      int i = 0;
      for (i = 0; i < n; i++)
          sum += *x++ * *y++;
      return sum;
}
int fib(int n)
{
      int r;
      if (n < 2) {
         r = 1;
      } else {
         int a = fib(n-1);
         int b = fib(n-2);
         r = a + b;
      }
      return r;
}
#pragma stack_bank(sysstack)
void count_ticks(void)
{
      extern int ticks;
      ticks++;
}
```

#pragma default_code_bank(*bankname*)

The default_code_bank pragma informs the compiler that *bankname* should be considered the default memory bank for the instructions generated for any following functions that do not explicitly use #pragma code_bank.

If *bankname* is omitted, the pragma sets the compiler's default back to not specifying a particular bank for generated code.

For more information, see Memory Bank Selection.

#pragma default_data_bank(*bankname*)

The `default_data_bank` pragma informs the compiler that *bankname* should be considered the default memory bank for non-local data accesses in any following functions that do not explicitly use `#pragma data_bank`.

If *bankname* is omitted, the pragma sets the compiler's default back to not specifying a particular bank for non-local data.

For more information, see Memory Bank Selection.

#pragma default_stack_bank(*bankname*)

The `default_stack_bank` pragma informs the compiler that *bankname* should be considered the default memory bank for local data in any following functions that do not explicitly use `#pragma stack_bank`.

If *bankname* is omitted, the pragma sets the compiler's default back to not specifying a particular bank for local data.

For more information, see Memory Bank Selection.

#pragma bank_memory_kind(*bankname, kind*)

The `bank_memory_kind` pragma informs the compiler of what *kind* of memory the memory bank *bankname* is. See Memory Bank Selection for the kinds supported by the compiler.

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition.

In the following example, the compiler knows that all accesses to the `data[]` array are to the "`blue`" memory bank, and hence to internal, in-core memory.

```
#pragma bank_memory_kind(blue, internal)
int sum_list(const int bank("blue") *data, int n)
{
      int sum = 0;
      while (n--)
          sum += data[n];
      return sum;
}
```

#pragma bank_read_cycles(*bankname, cycles[, bits]*)

The `bank_read_cycles` pragma tells the compiler that each read operation on the memory bank *bankname* requires *cycles* cycles before the resulting data is available. This allows the compiler to generate more efficient code.

If the `bits` parameter is specified, it indicates that a read of *bits* bits will take *cycles* cycles. If the *bits* parameter is omitted, the pragma indicates that reads of all widths will require *cycles* cycles. *bits* may be one of `32` or `64`.

In the following example, the compiler assumes that a read from `*x` takes a single cycle, as this is the default read time, but that a read from `*y` takes twenty cycles, because of the pragma.

```
#pragma bank_read_cycles(slowmem, 20)
int dotprod(int n, const int *x, bank("slowmem") const int *y)
{
      int i, sum;
      for (i=sum=0; i < n; i++)
          sum += *x++ * *y++;
      return sum;
}
```

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition.

#pragma bank_write_cycles(*bankname, cycles[, bits]*)

The `bank_write_cycles` pragma tells the compiler that each write operation on memory bank *`bankname`* requires *`cycles`* cycles before it completes. This allows the compiler to generate more efficient code.

If the `bits` parameter is specified, it indicates that a write of *`bits`* bits will take *`cycles`* cycles. If the *`bits`* parameter is omitted, the pragma indicates that writes of all widths will require *`cycles`* cycles. *`bits`* may be one of `32` or `64`.

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

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

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

#pragma bank_maximum_width(*bankname, width*)

The `bank_maximum_width` pragma informs the compiler that *`width`* is the maximum number of bits to transfer to/from memory bank *`bankname`* in a single access. On SHARC processors, the *`width`* parameter may only be `32` or `64`.

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition.

Code Generation Pragmas

The code generation pragmas are described below in the following sections.

#pragma avoid_anomaly_45 {on | off}

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

These pragmas, `#pragma avoid_anomaly_45 on` and `#pragma avoid_anomaly_45 off`, 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.

#pragma no_db_return

The pragma is used immediately before a function definition and will cause the compiler to ensure that non-delayed-branch instructions are used to return from the function. The pragma may be applied to both interrupt and non-interrupt function definitions. Applying the pragma to an interrupt function can be used as a workaround for ADSP-213xx silicon anomalies 02000069, 04000068, 06000028, 07000021, 08000026, and 09000015, "Incorrect Popping of stacks possible when exiting IRQx/Timer interrupts with DB modifiers."

If the pragma does not appear immediately before a function definition then a compiler error message is issued.

The following examples show uses of the pragma:

Example 1.

```
#pragma no_db_return
int max(int x, int y)
{
  if (y > x)
    return y;
  else
    return x;
}
```

Example 2.

```
#pragma no_db_return
#pragma interrupt_complete_nesting
void foo(void) {
...
}
```

Example 3.

```
#pragma no_db_return
int i;  /* INVALID - not a function definition,
           causes compiler error cc1943 */
```

Exceptions Table Pragma

The `generate_exceptions_tables` pragma allows you to generate code that permits exceptions to be thrown through C functions. It is described in the following section.

#pragma generate_exceptions_tables

The pragma may be applied to a C function definition to request the compiler to generate tables which enable C++ exceptions to be thrown through executions of this function.

The following example consists of two source files. The first is a C file which contains the pragma applied to the definition of function call_a_call_back.

```
#pragma generate_exceptions_tables
void call_a_call_back(void pfn(void)) {
   pfn();/* without pragma program terminates when
            throw_an_int throws an exception */
}
```

The second source file contains C++ code. The function main calls call_a_call_back, from the C file listed above, which in turn calls throw_an_int. The exception thrown by throw_an_int will be caught by the catch handler in main because use of the pragma ensured the compiler generated an exceptions table for call_a_call_back.

```
#include <iostream>
extern "C" void call_a_call_back(void pfn());

static void throw_an_int() {
   throw 3;
}

int main() {
   try {
     call_a_call_back(throw_an_int);
   } catch (int i) {
     if (i == 3) std::cout << "Test passed\n";
   }
}
```

An alternative to using #pragma generate_exceptions_tables is to compile C files with the -eh (enable exception handling) switch, which, for C files, is equivalent to using the pragma before every function definition.

Mixed Char-Size Interface Pragmas

This section describes pragmas that help with generating code that interfaces between code built with the -char-size-8 compiler switch and code built with the -char-size-32 compiler switch.

- #pragma byte_addressed [(push)|(pop)]
- #pragma word_addressed [(push)|(pop)]
- #pragma default_addressed [(push)|(pop)]
- #pragma no_stack_translation

#pragma byte_addressed [(push)|(pop)]

`#pragma byte_addressed` is used to specify that the following symbol(s) are defined in a source file built with the `-char-size-8` switch. It is generally used when creating an interface between a current source file, being compiled with the `-char-size-32` switch, and other source files built with the `-char-size-8` switch. The pragma may only be used on processors that support byte-addressing, shown in the ***Processors Supporting Byte-Addressing*** table (see Processor Features).

For a discussion of how to create interfaces between byte-addressed and word-addressed code using `#pragma byte_addressed`, see Mixed Char-Size Applications.

#pragma word_addressed [(push)|(pop)]

`#pragma word_addressed` is used to specify that the following symbol(s) are defined in a source file built with the `-char-size-32` switch. It is generally used when creating an interface between a current source file, being compiled with the `-char-size-8` switch, and other source files built with the `-char-size-32` switch. The pragma only has an effect on processors that support byte-addressing, shown in the ***Processors Supporting Byte-Addressing*** table (see Processor Features).

For a discussion of how to create interfaces between byte-addressed and word-addressed code using `#pragma word_addressed`, see Mixed Char-Size Applications.

#pragma default_addressed [(push)|(pop)]

`#pragma default_addressed` is used to specify that the following symbol(s) are defined in a source file built with the same `-char-size` switch as the current source file. It is generally used when creating a header file designed to be incorporated into either byte-addressed or word-addressed source files, where both word-addressed and byte-addressed versions of the declared symbols exist in a library. For example, the standard headers included in CCES, such as `stdio.h`, declare their symbols as `default_addressed` as both word-addressed and byte-addressed versions of standard library functions are included in the CCES standard libraries. The pragma overrides any `#pragma byte_addressed` or `#pragma word_addressed` that is currently in force. The pragma only has an effect on processors that support byte-addressing, shown in the ***Processors Supporting Byte-Addressing*** table (see Processor Features).

For a discussion of how to create interfaces between byte-addressed and word-addressed code, see Mixed Char-Size Applications.

#pragma no_stack_translation

`#pragma no_stack_translation` is used to specify that the following function declaration declares a function written in assembly code that will work correctly when the frame and stack pointers i6, i7, b6 and b7 are either in the word-addressed address space or in the byte-addressed address space. Therefore when calling this function from C/C++ code, the compiler does not need to translate the frame and stack pointers prior to and on return from the call. Using the pragma when it is valid to do so results in more efficient and smaller code. The pragma only has an effect on processors that support byte-addressing, shown in the ***Processors Supporting Byte-Addressing*** table (see Processor Features).

For a discussion of how to create interfaces between byte-addressed and word-addressed code using `#pragma no_stack_translation`, see Mixed Char-Size Applications.

GCC Compatibility Extensions

The compiler provides compatibility with many features of the C dialect accepted by version 4.6 of the GNU C Compiler. Many of these features are available in the ISO/IEC 9899:1999 standard. A brief description of the extensions is included in this section. For more information, refer to the following URL:

http://gcc.gnu.org/onlinedocs/gcc-4.6.4/gcc/index.html#toc_C-Extensions.

NOTE: The GCC compatibility extensions are only available in C89 and C99 modes. They are not accepted in C ++ dialect mode, unless enabled with the `-g++` switch.

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,

```
#define checker(p)      ({              \
   __label__ exit;                      \
   int i;                               \
   for (i=0; p[i]; ++i) {               \
      int d = get(p[i]);                \
      if (!check(d)) goto exit;         \
      process(d);                       \
   }                                    \
   exit:                                \
```

```
   i;                              \
   })

extern int g_p[100];
int checkit()  {
  int local_i = checker(g_p);
  return local_i;
}
```

NOTE: Statement expressions are not supported in C++ mode. Also, 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`.

NOTE: The `typeof` keyword is not supported in C++ mode. The `typeof` keyword is an extension to C originally implemented in the GCC compiler. It should be used with caution because it is not compatible with other dialects of C/C++ and has not been adopted by the more recent C99 standard.

Generalized Lvalues

Lvalues are expressions that may appear on the left-hand side of an assignment. GCC provides several lvalue-related extensions to C, which are supported by the compiler for GCC compatibility:

- A cast is an lvalue if its operand is an lvalue. This C-mode extension is not allowed in C++ mode.
- A comma operator is an lvalue if its right operand is an lvalue. This C-mode extension is a standard feature of C++.
- A conditional operator is an lvalue if its last two operands are lvalues of the same type. This C-mode extension 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.

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.

GCC Variable Argument Macros

The final parameter in a macro declaration may be followed by an ellipsis (`...`) to indicate the parameter stands for a variable number of arguments.

```
#define tracegcc(file,line,msg ...) \ logmsg(file,line, ## msg)
```

can be used with differing numbers of arguments: the following statements:

```
tracegcc("a.c", 999, "one", "two", "three");
tracegcc("a.c", 999, "one", "two");
tracegcc("a.c", 999, "one");
tracegcc("a.c", 999);
```

expand to the following code:

```
logmsg("a.c", 999,"one", "two", "three");
logmsg("a.c", 999,"one", "two");
logmsg("a.c", 999,"one");
logmsg("a.c", 999);
```

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.

NOTE: This variable argument macro syntax comes from GCC. The compiler supports GCC variable argument macro formats in C89 and C99 modes. In C++ mode the -g++-g++ switch needs to be enabled to support this feature. For more information on standard variable argument support, see Variable Argument Macros.

Line Breaks in String Literals

String literals may span many lines. The line breaks do not need to be escaped in any way. They are replaced by the character `\n` in the generated string. This extension is not supported in C++ mode. The extension is not compatible with many dialects of C, including ANSI/ISO C89 and C99. However, it is useful in `asm` statements, which are intrinsically non-portable.

This extension may be disabled via the `-no-multiline` switch.

Arithmetic on Pointers to Void and Pointers to Functions

Addition and subtraction is allowed on pointers to `void` and pointers to functions. The result is as if the operands had been cast to pointers to `char`. The `sizeof()` operator returns one for `void` and function types.

Cast to Union

A type cast can be used to create a value of a union type, by casting a value of one of the union's member types to the union type.

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 the three-period token `...`.

For example,

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

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*.

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`, or the label `N.` when compiling with the `-char-size-8` switch on processors that support byte-addressing.

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. The ***Keywords for __attribute__*** table lists the accepted keywords.

Table 2-44: Keywords for __attribute__

Attribute Keyword	*Behavior*
`alias("name")`	Accepted on functions declarations. Declares the function to be an alias for name.
`aligned(N)`	Accepted on variables, where it is equivalent to `#pragma align(N)`. Accepted (but ignored) on typedefs.
`always_inline`	Accepted on function declarations. Equivalent to the pragma of the same name.
`const`	Accepted on function declarations. Equivalent to the pragma of the same name.
`constructor`	Accepted (but ignored) on function declarations.
`deprecated`	Accepted on function, variable and type declarations. Causes the compiler to emit a warning if the entity with the attribute is referenced within the source code.
`destructor`	Accepted (but ignored) on function declarations.
`format(kind,str,args)`	Accepted on function declarations. Indicates that the function accepts a formatting argument string of type kind, e.g. `printf`. str and args are integer values; the strth parameter of the function is the formatting string, while the argsth parameter of the function is the first parameter processed by the formatting string.

Table 2-44: Keywords for __attribute__ (Continued)

Attribute Keyword	*Behavior*
`format_arg(kind, str)`	Accepted on function declarations. Indicates that the function accepts and returns a formatting argument string of type kind. str is an integer value; the strth parameter of the function is the formatting string.
`malloc`	Accepted on function declarations. Equivalent to using `#pragma alloc`.
`naked`	Accepted (but ignored) on function declarations.
`no_instrument_function`	Accepted (but ignored) on function declarations.
`nocommon`	Accepted on variable declarations. Causes the compiler to treat the declaration as a definition.
`noinline`	Accepted on function declarations. Equivalent to `#pragma never_inline`.
`nonnull`	Accepted on function declarations. Causes the compiler to emit a warning if the function is invoked with any NULL parameters.
`noreturn`	Accepted on function declarations. Equivalent to using the pragma of the same name.
`nothrow`	Accepted (but ignored) on function declarations.
`packed`	Accepted (but ignored) on typedefs. When used on variable declarations, this is equivalent to using the pragma of the same name.
`pure`	Accepted on function declarations. Equivalent to using the pragma of the same name.
`section("name")`	Accepted on function declarations. Equivalent to using the pragma of the same name.
`sentinel`	Accepted on function declarations. Directs the compiler to emit a warning for any calls to the function which do not provide a null pointer literal as the last parameter. Accepts an optional integer position P (default 0) to indicate that the Pth parameter from the end is the sentinel instead.
`transparent_union`	Accepted on union definitions. When the union type is used for a function's parameter, the parameter can accept values which match any of the union's types.
`unused`	Accepted on declarations of functions, variables and types. Indicates that the entity is known not to be used, so the compiler should not emit diagnostics complaining that there are no uses of the entity.
`used`	Accepted on declarations of functions and variables. Indicates that the compiler should emit the entity even when the compiler cannot detect uses. Similar to `#pragma retain_name`, but this attribute can be applied to static entities that will not be visible outside the module. Conversely, this attribute will not prevent linker elimination from deleting the entity.
`warn_unused_result`	Accepted (but ignored) on function declarations.
`weak`	Accepted on function and variable declarations. Equivalent to using `#pragma weak_entry`

Unnamed struct/union Fields Within struct/unions

The compiler allows you to define a structure or union that contains, as fields, structures and unions without names. For example:

```
struct {
   int field1;
   union {
```

```
        int field2;
        int field3;
    };
    int field4;
} myvar;
```

This allows the user to access the members of the unnamed union as though they were members of the enclosing struct; for example, `myvar.field2`.

Support for 40-Bit Arithmetic

The SHARC family of processors support 40-bit, floating-point arithmetic. Although this feature is not supported by the compiler, it is used by some run-time library functions and compiler support functions. This section provides information on the following topics:

- The implications of using 40-bit arithmetic in C/C++ code
- Library functions that use 40-bit arithmetic (directly or indirectly)

Using 40-Bit Arithmetic in Compiled Code

Although 40-bit arithmetic can be enabled in C/C++ code (by clearing the `RND32` bit on the MODE1 register), there are a number of factors that mean that arithmetic operations can produce inconsistent results:

- Where possible, the compiler will attempt to perform constant folding (the simplification of constant expressions at compilation time). The results of floating-point constant folding may be different from the results generated by performing the same calculation using the SHARC processor's 40-bit arithmetic.
- The compiler will sometimes use the integer `PASS` instruction (`"Rx = PASS Ry;"`) to copy a floating-point value from one register to another. This operation will result in a 40-bit value being truncated to a 32-bit value. It is not normally possible to predict whether the compiler will use this instruction-it depends on many factors, such as the code sequence being compiled and whether optimization has been enabled. However, using the `-extra-precision` switch prevents use of the integer `PASS` instruction for floating-point values. For more information, see `-extra-precision`.
- By default, data memory (including the stack) is configured to access 32-bit words, so any data stored to memory will be truncated from 40 bits to 32 bits. It is not possible to anticipate exactly when the compiler will place data in memory (especially when the optimizer has been enabled), meaning that it is not possible to guarantee that all 40 bits of a calculation will be preserved. For example, when preserving the value of a local variable across a function call, the compiler can either store the variable on the stack (which truncates it) or store it in a preserved register (for example, `R3` which will preserve all 40 bits). As before, the behavior depends on many factors such as the code sequence and optimization.

For these reasons, it is recommended that 40-bit arithmetic is not used in C/C++ code.

Run-Time Library Functions That Use 40-Bit Arithmetic

The following run-time library functions use 40-bit arithmetic:

cfft_mag	cosf	div	fir (the scalar-valued version from the header file filters.h)
iir (the scalar-valued version from the header file filters.h)	ldiv	fmodf	rfft_mag
rsqrtf	sinf	sqrtf	

The compiler support functions for the following operations use 40-bit arithmetic:

modulus operator	floating point division	integer division

A number of library functions do not themselves use 40-bit arithmetic but they invoke one or more of the above functions and may therefore generate less accurate results if they are interrupted:

acosf	asinf	cabsf	cartesianf
cexpf	normf	polarf	gen_blackman
gen_hamming	gen_hanning	gen_harris	gen_kaiser
rmsf	twidfftf	twiffft	

If the -double-size-64 switch has not been specified, then the following functions are also affected:

acos	asin	cabs	cartesian
cexp	cos	fmod	norm
polar	rms	rsqrt	sin
sqrt			

SIMD Support

The SHARC processors supported by the compiler allow Single Instruction, Multiple Data (SIMD) execution. When optimizing, the compiler can automatically exploit SIMD mode, subject to certain constraints being met. If the compiler is unable to automatically exploit SIMD mode, it will generate normal code (Single Instruction, Single Data, "SISD"). You can also use pragmas and other facilities to inform the compiler when SIMD mode is appropriate.

This section contains:

- A Brief Introduction to SIMD Mode
- What the Compiler Can Do Automatically
- What Prevents the Compiler From Automatically Exploiting SIMD Mode
- How to Help the Compiler Exploit SIMD Mode

- How to Prevent SIMD Code Generation

A Brief Introduction to SIMD Mode

This brief discussion is only concerned with aspects of SIMD architecture as they relate to the compiler. For full details on SIMD mode, refer to your processor's hardware reference manual.

In SIMD mode, the processor uses an additional computation unit operating in parallel with the first computation unit. This additional unit has its own register file. Whereas in SISD mode, only the first unit fetches values from memory, performs operations on them and stores the results back in memory, in SIMD mode, both units do this at once. The two units access adjacent memory locations, so that if the first unit accesses location *M*, the second unit will access location *M*+1. The operation performed in both units will be the same, but each unit will be performing the operation on its own data.

Because the processor is performing two operations in parallel, SIMD mode can provide double the computational throughput of SISD mode. However, because SIMD mode accesses adjacent memory locations, the compiler can only exploit SIMD mode when the source code being compiled supports such access patterns.

SIMD is a processor mode. For a given compute instruction *I*, if the processor is in SISD mode, the processor will execute *I* as a SISD instruction, on the first computation unit. In SIMD mode, the processor will execute the same instruction *I* as a SIMD instruction, executing it on both computation units. (Not all instructions behave differently in SISD and SIMD modes; for example, address arithmetic, which is not executed by the processor's computation unit, is not affected by the processor mode.)

What the Compiler Can Do Automatically

To exploit SIMD mode, you must enable the compiler optimizer.

No C/C++ language extensions are necessary for SIMD use; rather, the compiler can automatically generate SIMD code from standard C/C++ as long as there is sufficient information to indicate that the transformation does not alter the semantics of the source code.

There is a cost associated with switching between SISD and SIMD modes, so the compiler will generate code that exploits SIMD mode when it can determine that the source code meets the appropriate constraints and that the improvement in performance is likely to outweigh the cost of switching the modes.

Because of the cost of switching between modes, the compiler is most likely to generate SIMD code within loops, as the time spent within the loop in SIMD mode generally outweighs the mode-switching costs outside the loop. In contrast, SIMD mode is relatively rare in linear code, as the access patterns typically do not allow for many operations in a given mode before the mode must be switched again.

What Prevents the Compiler From Automatically Exploiting SIMD Mode

The compiler will verify that the source code is suitable for SIMD mode before transforming it. There are a number of reasons why a given piece of source code may not be suitable for SIMD mode, including:

- The memory access patterns are not suitable, e.g. they do not access adjacent memory locations.
- The number of consecutive operations that can exploit SIMD mode are insufficient to justify the cost of switch into SIMD and back to SISD.

- Some of the code is conditional, and the compiler cannot implement them with conditional instructions.
- The code contains function calls that cannot be inlined.
- The code is a loop that contains dependencies between successive iterations, i.e. there is an operation in iteration N+1 that depends on the result of an operation in iteration N.
- The code accesses memory locations that are not double-word aligned (ADSP-2116x processors only).
- The compiler cannot be certain that input and output buffer pointers do not point to the same array.
- The data being accessed might end up in external memory, and the target processor has external memory which does not support SIMD accesses, i.e. the target processor is one of ADSP-2116x, ADSP-2137x, ADSP-21367, ADSP-21368, or ADSP-21369.
- The code contains `asm` statements: the compiler has no knowledge of the instructions executed by `asm` statements, so cannot automatically determine whether the instructions would be safe in SIMD mode, unless you let the compiler know that an `asm` statement is safe for SIMD mode using the `-annotate-loop-instr` switch, described in `-asms-safe-in-simd-for-loops` .

When the compiler detects such problems, it automatically avoids using SIMD mode, and generates normal SISD mode code. The compiler will not generate SIMD code for source code where the compiler can determine that the resulting SIMD code would not be a valid representation of the source.

If the compiler determines that the *only* reason why the code is not suitable for SIMD mode is because the data in question is not appropriately aligned, it also issues a warning to that effect; you may be able to modify your source code so that the compiler can see that the data is suitably aligned, in which case SIMD mode code will be possible.

For more information, see How to Help the Compiler Exploit SIMD Mode.

If the compiler can neither prove that your code is suitable for SIMD nor prove that it is not, the compiler will take the conservative approach, and will generate SISD code.

How to Help the Compiler Exploit SIMD Mode

The Optimal Performance from C/C++ Source Code chapter contains a great deal of advice on how you can write your source code so that the compiler can obtain the information it needs to verify that SIMD mode is safe for your application.

The compiler will not automatically attempt to generate SIMD code if:

- the optimizer is not enabled.
- SIMD generation has been explicitly disabled; see How to Prevent SIMD Code Generation.
- the target processor supports external memory, and external memory does not support SIMD memory accesses.

Therefore, you should:

- Enable the optimizer; see Optimization Control.
- Remove any switches that disable SIMD generation.

- Specify the `-loop-simd` switch or the `-linear-simd` switch, as required, to tell the compiler to attempt SIMD generation, if your processor has external memory that is problematic for SIMD access.

This will mean that the compiler will attempt SIMD generation, but the compiler may still have difficulty if it cannot verify all the operations would be safe in SIMD mode.

In circumstances where your application is suitable for SIMD, but the compiler cannot prove this, the compiler will default to generating SISD code. In such cases, you can use #pragma SIMD_for. The pragma is used before a loop construct, and tells the compiler that for all memory accesses, you have verified that:

- where necessary the accesses are suitably aligned for SIMD mode.
- there are no accesses that rely on data stored during the previous iteration of the loop.
- the accesses do not alias each other.
- the code is not accessing external memory (if that is problematic for SIMD operations on the target processor).
- if there are any circular buffers used in the loop, the circular buffer length consists of an even number of elements, and the initial value of the pointer is aligned to point to an even element.

The pragma will not force the compiler to generate SIMD code if the compiler can prove that the source code is not suitable for SIMD mode, but where the compiler is unable to resolve the matter either way, #pragma SIMD_for tells the compiler that it is safe to proceed with SIMD mode code generation.

If your loop contains `asm` statements, this is normally a barrier to SIMD generation, since the compiler cannot tell whether the instructions within the `asm` statement would be valid in SIMD mode. However, you can specify the `-asms-safe-in-simd-for-loops` switch to tell the compiler that, for loops with the SIMD_for pragma, `asms` should be considered safe.

NOTE: SIMD accesses are not supported to external memory on ADSP-2116x, ADSP-2137x, ADSP-21367, ADSP-21368 or ADSP-21369 processors, so to obtain SIMD code for these processors, you must use #pragma SIMD_for, `-linear-simd`, or the `-loop-simd` switch to enable SIMD code generation, and you must ensure the associated data is not mapped to external memory.

How to Prevent SIMD Code Generation

To guarantee that the compiler does not generate SIMD code even when possible, use one of the following:

- The `-no-simd` switch
- `-no-simd`

You can disable SIMD generation for linear code, but still leave SIMD generation for loops enabled, using the `-no-linear-simd` switch.

Accessing External Memory on ADSP-2126x and ADSP-2136x Processors

On the ADSP-2126x and some ADSP-2136x processors, it is not possible to access external memory directly from the processor core. The compiler provides some facilities to allow access to variables in external memory from C/C++ code, and to reduce the possibility of errors due to incorrect data placement.

Link-Time Checking of Data Placement

Data which is placed in external memory on ADSP-2126x and 2136x processors must be defined using the `DMAONLY` qualifier of the `section` or `default_section` pragmas (#pragma section/#pragma default_section). For example:

```
#pragma section("seg_extmem1", DMAONLY)
int extmem1[100];
```

The linker will perform additional checks to ensure that data marked as `DMAONLY` is not placed in internal memory, and that "normal" data is not placed in external memory. If data is placed incorrectly, the linker will issue an error.

Refer to the *Linker and Utilities Manual* for information for more information.

Inline Functions for External Memory Access

Two inline functions, `read_extmem` and `write_extmem`, are provided to transfer data between internal and external memory. A full description of these functions is provided in the *C/C++ Library Manual for SHARC Processors*.

Preprocessor Features

C/C++ preprocessor are used by CCES to control the programming environment. The `cc21k` compiler provides standard preprocessor functionality, as described in any C text. The following extensions to standard C are also supported including several features of the

- Predefined Preprocessor Macros
- Preprocessor Generated Warnings
- GCC Variable Argument Macros

NOTE: The compiler's preprocessor is an integral part of the compiler; it is not the preprocessor described in the *Assembler and Preprocessor Manual*.

This section contains:

- Predefined Preprocessor Macros
- Writing Macros

Predefined Preprocessor Macros

The ***Predefined Preprocessor Macro Listing*** table describes the predefined preprocessor macros.

cc21k defines a hierarchy of macros that can be used in code to identify the processor for which you are compiling. These macros identify the processor by:

- Architecture, to identify the processor as ADSP-21xxx or ADSP-SCxxx (by the __ADSP21000__ macro)
- The Instruction Set Architecture (ISA) revision supported (via the __ADSPSHARC__ macro)
- Processor family (e.g. the __ADSP214xx__, __ADSP2146x__ macros)
- Family that use a common library build (e.g. the __ADSP21469_FAMILY__ macro)
- Processor itself (e.g. the __ADSP21467__ macro)

Table 2-45: Predefined Preprocessor Macro Listing

Macro	*Function*
__2116x__	When compiling for the ADSP-21160 or ADSP-21161 processors, cc21k defines __2116x__ as 1. This macro has been deprecated; please use the __ADSP2116x__ macro instead.
__2126x__	When compiling for the ADSP-21261, ADSP-21262, or ADSP-21266 processors, cc21k defines __2126x__ as 1. This macro has been deprecated; please use the __ADSP2126x__ macro instead.
__213xx__	When compiling for the ADSP-21362, ADSP-21363, ADSP-21364, ADSP-21365, ADSP-21366, ADSP-21367, ADSP-21368, ADSP-21369, ADSP-21371 or ADSP-21375 processors, cc21k defines __2136x__ and __213xx__ as 1. This macro has been deprecated; please use the __ADSP213xx__ macro instead.
__2136x__	When compiling for the ADSP-21362, ADSP-21363, ADSP-21364, ADSP-21365, ADSP-21366, ADSP-21367, ADSP-21368 or ADSP-21369 processors, cc21k defines __2136x__ as 1. This macro has been deprecated; please use the __ADSP2136x__ macro instead.
__2137x__	When compiling for the ADSP-21371 and ADSP-21375 processors, cc21k defines __2137x__ as 1. This macro has been deprecated; please use the __ADSP2137x__ macro instead.
__214xx__	When compiling for the ADSP-2146x, ADSP-2147x or ADSP-2148x processors, cc21k defines __214xx__ as 1. This macro has been deprecated; please use the __ADSP214xx__ macro instead.
__2146x__	When compiling for the ADSP-21467 or ADSP-21469 processors, cc21k defines __2146x__ as 1. This macro has been deprecated; please use the __ADSP2146x__ macro instead.
__2147x__	When compiling for the ADSP-21477, ADSP-21478, or ADSP-21479 processors, cc21k defines __2147x__ as 1. This macro has been deprecated; please use the __ADSP2147x__ macro instead.
__2148x__	When compiling for the ADSP-21483, ADSP-21486, ADSP-21487, ADSP-21488, or ADSP-21489 processors, cc21k defines __2148x__ as 1. This macro has been deprecated; please use the __ADSP2148x__ macro instead.

Table 2-45: Predefined Preprocessor Macro Listing (Continued)

Macro	*Function*
`__ADSP21000__`	`cc21k` always defines `__ADSP21000__` as 1.
`__ADSP21160__`	`cc21k` defines `__ADSP21160__` as 1 when you compile with the `-proc ADSP-21160` command-line switch. When compiling for the ADSP-21160 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21160_FAMILY__`, `__ADSP2116x__`, `__ADSP211xx__` and `__SIMDSHARC__`.
`__ADSP21160_FAMILY__`	When compiling for the ADSP-21160 processor, `cc21k` defines `__ADSP21160_FAMILY__` as 1.
`__ADSP21161__`	`cc21k` defines `__ADSP21161__` as 1 when you compile with the `-proc ADSP-21161` command-line switch. When compiling for the ADSP-21161 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21161_FAMILY__`, `__ADSP2116x__`, `__ADSP211xx__` and `__SIMDSHARC__`.
`__ADSP21161_FAMILY__`	When compiling for the ADSP-21161 processor, `cc21k` defines `__ADSP21161_FAMILY__` as 1.
`__ADSP2116x__`	When compiling for the ADSP-21160 or ADSP-21161 processors, `cc21k` defines `__ADSP2116x__` as 1.
`__ADSP211xx__`	When compiling for the ADSP-21160 or ADSP-21161 processors, `cc21k` defines `__ADSP211xx__` as 1.
`__ADSP21261__`	`cc21k` defines `__ADSP21261__` as 1 when you compile with the `-proc ADSP-21261` command-line switch. When compiling for the ADSP-21261 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21266_FAMILY__`, `__ADSP2126x__`, `__ADSP212xx__`, and `__SIMDSHARC__`.
`__ADSP21262__`	`cc21k` defines `__ADSP21262__` as 1 when you compile with the `-proc ADSP-21262` command-line switch. When compiling for the ADSP-21262 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21266_FAMILY__`, `__ADSP2126x__`, `__ADSP212xx__`, and `__SIMDSHARC__`.
`__ADSP21266__`	`cc21k` defines `__ADSP21266__` as 1 when you compile with the `-proc ADSP-21266` command-line switch. When compiling for the ADSP-21266 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21266_FAMILY__`, `__ADSP2126x__`, `__ADSP212xx__`, and `__SIMDSHARC__`.
`__ADSP21266_FAMILY__`	When compiling for the ADSP-21261, ADSP-21262 or ADSP-21266 processors, `cc21k` defines `__ADSP21266_FAMILY__` as 1.
`__ADSP2126x__`	When compiling for the ADSP-21261, ADSP-21262 or ADSP-21266 processors, `cc21k` defines `__ADSP2126x__` as 1.

Table 2-45: Predefined Preprocessor Macro Listing (Continued)

Macro	*Function*
__ADSP212xx__	When compiling for the ADSP-21261, ADSP-21262 or ADSP-21266 processors, cc21k defines __ADSP212xx__ as 1.
__ADSP21362__	cc21k defines __ADSP21362__ as 1 when you compile with the -proc ADSP-21362 command-line switch. When compiling for the ADSP-21362 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSP21362_FAMILY__, __ADSP2136x__, __ADSP213xx__ and __SIMDSHARC__.
__ADSP21362_FAMILY__	When compiling for the ADSP-21362, ADSP-21363, ADSP-21364, ADSP-21365 or ADSP-21366 processors, cc21k defines __ADSP21362_FAMILY__ as 1.
__ADSP21363__	cc21k defines __ADSP21363__ as 1 when you compile with the -proc ADSP-21363 command-line switch. When compiling for the ADSP-21363 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSP21362_FAMILY__, __ADSP2136x__, __ADSP213xx__ and __SIMDSHARC__.
__ADSP21364__	cc21k defines __ADSP21364__ as 1 when you compile with the -proc ADSP-21364 command-line switch. When compiling for the ADSP-21364 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSP21362_FAMILY__, __ADSP2136x__, __ADSP213xx__ and __SIMDSHARC__.
__ADSP21365__	cc21k defines __ADSP21365__ as 1 when you compile with the -proc ADSP-21365 command-line switch. When compiling for the ADSP-21365 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSP21362_FAMILY__, __ADSP2136x__, __ADSP213xx__ and __SIMDSHARC__.
__ADSP21366__	cc21k defines __ADSP21366__ as 1 when you compile with the -proc ADSP-21366 command-line switch. When compiling for the ADSP-21366 processors, the following additional macros are defined as 1: __ADSP21000__, __ADSP21362_FAMILY__, __ADSP2136x__, __ADSP213xx__ and __SIMDSHARC__.
__ADSP21367__	cc21k defines __ADSP21367__ as 1 when you compile with the -proc ADSP-21367 command-line switch. When compiling for the ADSP-21367 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSP21367_FAMILY__, __ADSP2136x__, __ADSP213xx__ and __SIMDSHARC__.
__ADSP21367_FAMILY__	When compiling for the ADSP-21367, ADSP-21368 or ADSP-21369 processors, cc21k defines __ADSP21367_FAMILY__ as 1.

Table 2-45: Predefined Preprocessor Macro Listing (Continued)

Macro	*Function*
`__ADSP21368__`	`cc21k` defines `__ADSP21368__` as 1 when you compile with the `-proc ADSP-21368` command-line switch. When compiling for the ADSP-21368 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21367_FAMILY__`, `__ADSP2136x__`, `__ADSP213xx__` and `__SIMDSHARC__`.
`__ADSP21369__`	`cc21k` defines `__ADSP21369__` as 1 when you compile with the `-proc ADSP-21369` command-line switch. When compiling for the ADSP-21369 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21367_FAMILY__`, `__ADSP2136x__`, `__ADSP213xx__` and `__SIMDSHARC__`.
`__ADSP2136x__`	When compiling for the ADSP-21362, ADSP-21363, ADSP-21364, ADSP-21365, ADSP-21366, ADSP-21367, ADSP-21368 or ADSP-21369 processors, `cc21k` defines `__ADSP2136x__` as 1.
`__ADSP21371__`	`cc21k` defines `__ADSP21371__` as 1 when you compile with the `-proc ADSP-21371` command-line switch. When compiling for the ADSP-21371 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21371_FAMILY__`, `__ADSP2137x__`, `__ADSP213xx__` and `__SIMDSHARC__`.
`__ADSP21371_FAMILY__`	When compiling for the ADSP-21371 or ADSP-21375 processors, `cc21k` defines `__ADSP21371_FAMILY__` as 1.
`__ADSP21375__`	`cc21k` defines `__ADSP21375__` as 1 when you compile with the `-proc ADSP-21375` command-line switch. When compiling for the ADSP-21375 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21371_FAMILY__`, `__ADSP2137x__`, `__ADSP213xx__` and `__SIMDSHARC__`.
`__ADSP2137x__`	When compiling for the ADSP-21371 or ADSP-21375 processors, `cc21k` defines `__ADSP2137x__` as 1.
`__ADSP213xx__`	When compiling for the ADSP-21362, ADSP-21363, ADSP-21364, ADSP-21365, ADSP-21366, ADSP-21367, ADSP-21368, ADSP-21369, ADSP-21371, or ADSP-21375 processors, `cc21k` defines `__ADSP213xx__` as 1.
`__ADSP21467__`	`cc21k` defines `__ADSP21467__` as 1 when you compile with the `-proc ADSP-21467` command-line switch. When compiling for the ADSP-21467 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21469_FAMILY__`, `__ADSP2146x__`, `__ADSP214xx__` and `__SIMDSHARC__`.
`__ADSP21469__`	`cc21k` defines `__ADSP21469__` as 1 when you compile with the `-proc ADSP-21469` command-line switch. When compiling for the ADSP-21469 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21469_FAMILY__`, `__ADSP2146x__`, `__ADSP214xx__` and `__SIMDSHARC__`.

Table 2-45: Predefined Preprocessor Macro Listing (Continued)

Macro	*Function*
__ADSP21469_FAMILY__	When compiling for the ADSP-21467 or ADSP-21469 processors, cc21k defines __ADSP21469_FAMILY__ as 1.
__ADSP2146x__	When compiling for the ADSP-21467 or ADSP-21469 processors, cc21k defines __ADSP2146x__ as 1.
__ADSP21477__	cc21k defines __ADSP21477__ as 1 when you compile with the -proc ADSP-21477 command-line switch. When compiling for the ADSP-21477 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSP21479_FAMILY__, __ADSP2147x__, __ADSP214xx__ and __SIMDSHARC__.
__ADSP21478__	cc21k defines __ADSP21478__ as 1 when you compile with the -proc ADSP-21478 command-line switch. When compiling for the ADSP-21478 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSP21479_FAMILY__, __ADSP2147x__, __ADSP214xx__ and __SIMDSHARC__.
__ADSP21479__	cc21k defines __ADSP21479__ as 1 when you compile with the -proc ADSP-21479 command-line switch. When compiling for the ADSP-21479 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSP21479_FAMILY__, __ADSP2147x__, __ADSP214xx__ and __SIMDSHARC__.
__ADSP21479_FAMILY__	When compiling for the ADSP-21477, ADSP-21478, ADSP-21479, ADSP-21483, ADSP-21486, ADSP-21487, ADSP-21488 or ADSP-21489 processors, cc21k defines __ADSP21479_FAMILY__ as 1.
__ADSP2147x__	When compiling for the ADSP-21477. ADSP-21478 or ADSP-21479 processors, cc21k defines __ADSP2147x__ as 1.
__ADSP21483__	cc21k defines __ADSP21483__ as 1 when you compile with the -proc ADSP-21483 command-line switch. When compiling for the ADSP-21483 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSP21479_FAMILY__, __ADSP2148x__, __ADSP214xx__ and __SIMDSHARC__.
__ADSP21486__	cc21k defines __ADSP21486__ as 1 when you compile with the -proc ADSP-21486 command-line switch. When compiling for the ADSP-21486 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSP21479_FAMILY__, __ADSP2148x__, __ADSP214xx__ and __SIMDSHARC__.
__ADSP21487__	cc21k defines __ADSP21487__ as 1 when you compile with the -proc ADSP-21487 command-line switch. When compiling for the ADSP-21487 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSP21479_FAMILY__, __ADSP2148x__, __ADSP214xx__ and __SIMDSHARC__.

Table 2-45: Predefined Preprocessor Macro Listing (Continued)

Macro	*Function*
`__ADSP21488__`	`cc21k` defines `__ADSP21488__` as 1 when you compile with the `-proc ADSP-21488` command-line switch. When compiling for the ADSP-21488 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21479_FAMILY__`, `__ADSP2148x__`, `__ADSP214xx__`, and `__SIMDSHARC__`.
`__ADSP21489__`	`cc21k` defines `__ADSP21489__` as 1 when you compile with the `-proc ADSP-21489` command-line switch. When compiling for the ADSP-21489 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21479_FAMILY__`, `__ADSP2148x__`, `__ADSP214xx__` and `__SIMDSHARC__`.
`__ADSP2148x__`	When compiling for the ADSP-21483, ADSP-21486, ADSP-21487, ADSP-21488 or ADSP-21489 processors, `cc21k` defines `__ADSP2148x__` as 1.
`__ADSP214xx__`	When compiling for the ADSP-21467, ADSP-21469, ADSP-21477, ADSP-21478, ADSP-21479, ADSP-21483, ADSP-21486, ADSP-21487, ADSP-21488, or ADSP-21489 processors, `cc21k` defines `__ADSP214xx__` as 1.
`__ADSP21562__`	`cc21k` defines `__ADSP21562__` as 1 when you compile with the `-proc ADSP-21562` command-line switch. When compiling for the ADSP-21562 processor the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21569_FAMILY__`, `__ADSP2156x__`, `__ADSP215xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSP21563__`	`cc21k` defines `__ADSP21563__` as 1 when you compile with the `-proc ADSP-21563` command-line switch. When compiling for the ADSP-21563 processor the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21569_FAMILY__`, `__ADSP2156x__`, `__ADSP215xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSP21565__`	`cc21k` defines `__ADSP21565__` as 1 when you compile with the `-proc ADSP-21565` command-line switch. When compiling for the ADSP-21565 processor the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21569_FAMILY__`, `__ADSP2156x__`, `__ADSP215xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSP21566__`	`cc21k` defines `__ADSP21566__` as 1 when you compile with the `-proc ADSP-21566` command-line switch. When compiling for the ADSP-21566 processor the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21569_FAMILY__`, `__ADSP2156x__`, `__ADSP215xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSP21567__`	`cc21k` defines `__ADSP21567__` as 1 when you compile with the `-proc ADSP-21567` command-line switch. When compiling for the ADSP-21567 processor the following additional macros are defined as 1: `__ADSP21000__`, `__ADSP21569_FAMILY__`, `__ADSP2156x__`, `__ADSP215xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.

Table 2-45: Predefined Preprocessor Macro Listing (Continued)

Macro	*Function*
__ADSP21569__	cc21k defines __ADSP21569__ as 1 when you compile with the -proc ADSP-21569 command-line switch. When compiling for the ADSP-21569 processor the following additional macros are defined as 1: __ADSP21000__, __ADSP21569_FAMILY__, __ADSP2156x__, __ADSP215xx__, __SIMDSHARC__ and __BA_SHARC__.
__ADSP21569_FAMILY__	When compiling for the ADSP-21562, ADSP-21563, ADSP-21565, ADSP-21566, ADSP-21567 or ADSP-21569 processors, cc21k defines __ADSP21569_FAMILY__ as 1.
__ADSP21571__	cc21k defines __ADSP21571__ as 1 when you compile with the -proc ADSP-21571 command-line switch. When compiling for the ADSP-21571 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSPSC573_FAMILY__, __ADSP2157x__, __ADSP215xx__, __SIMDSHARC__ and __BA_SHARC__.
__ADSP21573__	cc21k defines __ADSP21573__ as 1 when you compile with the -proc ADSP-21573 command-line switch. When compiling for the ADSP-21573 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSPSC573_FAMILY__, __ADSP2157x__, __ADSP215xx__, __SIMDSHARC__ and __BA_SHARC__.
__ADSP21583__	cc21k defines __ADSP21583__ as 1 when you compile with the -proc ADSP-21584 command-line switch. When compiling for the ADSP-21583 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSPSC583_FAMILY__, __ADSP2158x__, __ADSP215xx__, __SIMDSHARC__ and __BA_SHARC__.
__ADSP21584__	cc21k defines __ADSP21584__ as 1 when you compile with the -proc ADSP-21584 command-line switch. When compiling for the ADSP-21584 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSPSC589_FAMILY__, __ADSP2158x__, __ADSP215xx__, __SIMDSHARC__ and __BA_SHARC__.
__ADSP21587__	cc21k defines __ADSP21587__ as 1 when you compile with the -proc ADSP-21587 command-line switch. When compiling for the ADSP-21587 processor, the following additional macros are defined as 1: __ADSP21000__, __ADSPSC589_FAMILY__, __ADSP2158x__, __ADSP215xx__, __SIMDSHARC__ and __BA_SHARC__.
__ADSP2156x__	When compiling for the ADSP-21562, ADSP-21563, ADSP-21565, ADSP-21566, ADSP-21567 or ADSP-21569 processors, cc21k defines __ADSP2156x__ as 1.
__ADSP2157x__	When compiling for the ADSP-21571, ADSP-21573, ADSP-SC570, ADSP-SC571, ADSP-SC572 or ADSP-SC573 processors, cc21k defines __ADSP2157x__ as 1.
__ADSP2158x__	When compiling for the ADSP-21583, ADSP-21584, ADSP-21587, ADSP-SC582, ADSP-SC583, ADSP-SC584, ADSP-SC587 or ADSP-SC589 processors, cc21k defines __ADSP2158x__ as 1.

Table 2-45: Predefined Preprocessor Macro Listing (Continued)

Macro	*Function*
`__ADSP215xx__`	When compiling for the ADSP-21562, ADSP-21563, ADSP-21565, ADSP-21566, ADSP-21567, ADSP-21569, ADSP-21571, ADSP-21573, ADSP-21583, ADSP-21584, ADSP-21587, ADSP-SC570, ADSP-SC571, ADSP-SC572, ADSP-SC573, ADSP-SC582, ADSP-SC583, ADSP-SC584, ADSP-SC587 or ADSP-SC589 processors, `cc21k` defines `__ADSP215xx__` as 1.
`__ADSPSC570__`	`cc21k` defines `__ADSPSC570__` as 1 when you compile with the `-proc ADSP-SC570` command-line switch. When compiling for the ADSP-SC570 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSPSC573_FAMILY__`, `__ADSP2157x__`, `__ADSP215xx__`, `__ADSPSC57x__`, `__ADSPSC5xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSPSC571__`	`cc21k` defines `__ADSPSC571__` as 1 when you compile with the `-proc ADSP-SC571` command-line switch. When compiling for the ADSP-SC571 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSPSC573_FAMILY__`, `__ADSP2157x__`, `__ADSP215xx__`, `__ADSPSC57x__`, `__ADSPSC5xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSPSC572__`	`cc21k` defines `__ADSPSC572__` as 1 when you compile with the `-proc ADSP-SC572` command-line switch. When compiling for the ADSP-SC570 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSPSC573_FAMILY__`, `__ADSP2157x__`, `__ADSP215xx__`, `__ADSPSC57x__`, `__ADSPSC5xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSPSC573__`	`cc21k` defines `__ADSPSC573__` as 1 when you compile with the `-proc ADSP-SC573` command-line switch. When compiling for the ADSP-SC573 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSPSC573_FAMILY__`, `__ADSP2157x__`, `__ADSP215xx__`, `__ADSPSC57x__`, `__ADSPSC5xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSPSC582__`	`cc21k` defines `__ADSPSC582__` as 1 when you compile with the `-proc ADSP-SC582` command-line switch. When compiling for the ADSP-SC582 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSPSC589_FAMILY__`, `__ADSP2158x__`, `__ADSP215xx__`, `__ADSPSC58x__`, `__ADSPSC5xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSPSC583__`	`cc21k` defines `__ADSPSC583__` as 1 when you compile with the `-proc ADSP-SC583` command-line switch. When compiling for the ADSP-SC583 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSPSC589_FAMILY__`, `__ADSP2158x__`, `__ADSP215xx__`, `__ADSPSC58x__`, `__ADSPSC5xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.

Table 2-45: Predefined Preprocessor Macro Listing (Continued)

Macro	*Function*
`__ADSPSC584__`	`cc21k` defines `__ADSPSC584__` as 1 when you compile with the `-proc ADSP-SC584` command-line switch. When compiling for the ADSP-SC584 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSPSC589_FAMILY__`, `__ADSP2158x__`, `__ADSP215xx__`, `__ADSPSC58x__`, `__ADSPSC5xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSPSC587__`	`cc21k` defines `__ADSPSC587__` as 1 when you compile with the `-proc ADSP-SC587` command-line switch. When compiling for the ADSP-SC587 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSPSC589_FAMILY__`, `__ADSP2158x__`, `__ADSP215xx__`, `__ADSPSC58x__`, `__ADSPSC5xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSPSC589__`	`cc21k` defines `__ADSPSC589__` as 1 when you compile with the `-proc ADSP-SC589` command-line switch. When compiling for the ADSP-SC589 processor, the following additional macros are defined as 1: `__ADSP21000__`, `__ADSPSC589_FAMILY__`, `__ADSP2158x__`, `__ADSP215xx__`, `__ADSPSC58x__`, `__ADSPSC5xx__`, `__SIMDSHARC__` and `__BA_SHARC__`.
`__ADSPSC573_FAMILY__`	When compiling for the ADSP-21571, ADSP-21573, ADSP-SC570, ADSP-SC571, ADSP-SC572 or ADSP-SC573 processors, `cc21k` defines `__ADSPSC573_FAMILY__` as 1.
`__ADSPSC589_FAMILY__`	When compiling for the ADSP-21583, ADSP-21584, ADSP-21587, ADSP-SC582, ADSP-SC583, ADSP-SC584, ADSP-SC587 or ADSP-SC589 processors, `cc21k` defines `__ADSPSC589_FAMILY__` as 1.
`__ADSPSC57x__`	When compiling for the ADSP-SC570, ADSP-SC571, ADSP-SC572 or ADSP-SC573 processors, `cc21k` defines `__ADSPSC57x__` as 1.
`__ADSPSC58x__`	When compiling for the ADSP-SC582, ADSP-SC583, ADSP-SC584, ADSP-SC587 or ADSP-SC589 processors, `cc21k` defines `__ADSPSC58x__` as 1.
`__ADSPSC5xx__`	When compiling for the ADSP-SC50, ADSP-SC571, ADSP-SC572, ADSP-SC573, ADSP-SC582, ADSP-SC583, ADSP-SC584, ADSP-SC587 or ADSP-SC589 processors, `cc21k` defines `__ADSPSC5xx__` as 1.
`__ADSPSHARC__`	`cc21k` defines `__ADSPSHARC__` when compiling for any ADSP-21xxx or ADSP-SCxxx processor. The value represents the revision of the ISA supported by the processor, according to the following list: ADSP-211xx/212xx/213xx processors: 0x110, ADSP-2146x processors: 0x140, ADSP-2147x/2148x processors: 0x147, ADSP-2158x/SC58x processors: 0x200, ADSP-2157x/SC57x: 0x210, ADSP-2156x: 0x220.
`_ADI_COMPILER`	Always defined as 1.
`__ANALOG_EXTENSIONS__`	`cc21k` defines `__ANALOG_EXTENSIONS__` as 1 unless MISRA-C is enabled.
`_ADI_THREADS`	Defined as 1 when compiling with the `-threads` switch.

Table 2-45: Predefined Preprocessor Macro Listing (Continued)

Macro	*Function*
`__BASE_FILE__`	The preprocessor expands this macro to a string constant which is the current source file being compiled as seen on the compiler command-line.
`__BA_SHARC__`	`cc21k` defines `__BA_SHARC__` as 1 when compiling for processors that support byte-addressing (see the ***Processors Supporting Byte-Addressing*** table in Processor Features). The `__BA_SHARC__` macro is used to identify processors that are capable of byte-addressing independent of whether the `-char-size-8` or `-char-size-32` switch is being used.
`__BYTE_ADDRESSING__`	`cc21k` defines `__BYTE_ADDRESSING__` as 1 when the `-char-size-8` switch is used on processors that support byte-addressing (see the ***Processors Supporting Byte-Addressing*** table in Processor Features). When the `-char-size-32` compiler switch is used (`-char-size[-8|-32]`), the macro is not defined.
`__CCESVERSION__`	The preprocessor defines this macro to be an eight-digit hexadecimal representation of the CCES release, in the form `0xMMmmUUPP`, where:- `MM` is the major release number- `mm` is the minor release number- `UU` is the update number- `PP` is the patch release number For example, CrossCore Embedded Studio 1.0.2.0 would define __CCESVERSION__ as `0x01000200`.
`__cplusplus`	`cc21k` defines `__cplusplus` as 199711L when compiling C++ source files. It also gets defined as 1 for use in LDFs.
`__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 `-double-size-64` compiler switch is used (`-double-size[-32|-64]`), 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 compiler 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.
`__EXCEPTIONS`	`cc21k` defines `__EXCEPTIONS` as 1 when C++ exception handling is enabled (using the `-eh` command-line switch).
`__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).
`__FIXED_POINT_ALLOWED`	Defined as 1 unless MISRA-C is enabled. It is defined to indicate that the native fixed-point types support may be used. For more information, see Using Native Fixed-Point Types.
`__FLT64_SHARC__`	`cc21k` defines `__FLT64_SHARC__` as 1 when compiling for processors that support native double-precision floating-point arithmetic (see the ***Processors Supporting Native Double-Precision Floating-Point Arithmetic*** table in Processor Features).
`_HEAP_DEBUG`	`cc21k` defines `_HEAP_DEBUG` as 1 when Heap Debugging support is enabled, otherwise `_HEAP_DEBUG` is undefined. For more information, see Heap Debugging.

Table 2-45: Predefined Preprocessor Macro Listing (Continued)

Macro	*Function*
__HETEROGENEOUS_PROCESSOR__	cc21k defines __HETEROGENEOUS_PROCESSOR__ as 1 when the target processor comprises both SHARC and ARM cores (specifically, the ADSP-SC5xx family of processors), otherwise __HETEROGENEOUS_PROCESSOR__ is undefined.
__IDENT__	The preprocessor expands __IDENT__ to a string normally set using #ident.
_INSTRUMENTED_PROFILING	cc21k defines _INSTRUMENTED_PROFILING as 1 when instrumented profiling is enabled (using the -p switch).
_LANGUAGE_C	cc21k always defines _LANGUAGE_C as 1 when building C or C++ sources.
__LINE__	The preprocessor expands this macro into the current input line number as a decimal integer constant (ANSI standard).
_LONG_LONG	cc21k always defines _LONG_LONG as 1 when compiling C and C++ sources to indicate that 64-bit double word integer types are supported.
_MISRA_RULES	cc21k defines _MISRA_RULES as 1 when compiling in MISRA-C mode.
__NOSIMD__	cc21k defines __NOSIMD__ as 1 when SIMD code generation is disabled (using the -no-simd command-line switch).
__NORMAL_WORD_CODE__	When compiling for ADSP-214xx processors, cc21k defines __NORMAL_WORD_CODE__ as 1, when compiling in normal-word mode. For more information, see -nwc.
__NUM_ARM_CORES__	cc21k defines __NUM_ARM_CORES__ to the number of ARM cores on the target part. For ADSP-211xx/212xx/213xx/214xx/215xx processors, this is undefined; for ADSP-SC5xx processors, this is defined to 1.
__NUM_CORES__	cc21k defines __NUM_CORES__ to the total number of cores on the target part. For ADSP-211xx/212xx/213xx/214xx processors, this is always 1; for ADSP-SC5xx processors except ADSP-SC570, ADSP-SC572 and ADSP-SC582, this is defined to 3; for ADSP-SC570, ADSP-SC572, ADSP-SC582 and ADSP-215xx processors, this is defined to 2.
__NUM_SHARC_CORES__	cc21k defines __NUM_SHARC_CORES__ to the total number of SHARC cores on the target part. For ADSP-211xx/212xx/213xx/214xx processors, this macro is not defined; for ADSP-215xx and ADSP-SC5xx processors except ADSP-SC570, ADSP-SC572 and ADSP-SC582, this is defined to 2; for ADSP-SC570, ADSP-SC572 and ADSP-SC582 this is defined to 1.
_PGO_HW	cc21k defines _PGO_HW as 1 when you compile with the -pguide and -prof-hw command-line switches.
__RTTI	cc21k defines __RTTI as 1 when C++ run-time type information is enabled (using the -rtti command-line switch -pguide).
__SHORT_WORD_CODE__	When compiling for ADSP-214xx, ADSP-215xx and ADSP-SC5xx processors, cc21k defines __SHORT_WORD_CODE__ as 1, when compiling in short-word mode. This is the default when compiling for these processors.
__SIGNED_CHARS__	cc21k defines __SIGNED_CHARS__ as 1 indicating that plain char type variables are signed rather than unsigned. The macro is defined by default.

Table 2-45: Predefined Preprocessor Macro Listing (Continued)

Macro	*Function*
`__SILICON_REVISION__`	`cc21k` defines `__SILICON_REVISION__` to a hexadecimal constant corresponding to the target processor revision. For more information, see Using the `-si-revision` Switch switch.
`__SIMDSHARC__`	`cc21k` defines `__SIMDSHARC__` as 1. The `__SIMDSHARC__` macro is used to identify processors that are capable of executing SIMD code.
`__STDC__`	`cc21k` always defines `__STDC__` as 1.
`__STDC_VERSION__`	`cc21k` defines `__STDC_VERSION__` as 199409L when compiling in C89 mode, and as 199901L when compiling in C99 mode.
`__TIME__`	The preprocessor expands this macro into the preprocessing time as a string constant. The date string constant takes the form `hh:mm:ss` (ANSI standard).
`__VERSION__`	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 0x07010000 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` *`version`* switch has a value other than "none" or if any specific workaround is selected by means of the `-workaround` compiler switch (`-workaround` *`workaround_id[, workaround_id…]`*).

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 Macros

Whenever possible, use inline functions rather than compound macros. If compound macros are necessary, define such macros to allow invocation like function calls. This will make your source code easier to read and maintain. If you want your macro to extend over more than one line, you must escape the newlines with backslashes. If your macro contains a string literal and you are using the `-no-multiline` switch, then you must escape the newline twice, once for the macro and once for the string.

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 SHARC processor C/C++ run-time model and run-time environment. The C/C++ run-time model, which applies to compiler-generated code, includes descriptions of layout of the stack, data access, and call/entry sequence. The C/C++ run-time environment includes the conventions that C/C++ routines must follow to run on SHARC processors. Assembly routines linked to C/C++ routines must follow these conventions.

NOTE: Analog Devices recommends that assembly programmers maintain stack conventions.

The run-time environment issues include the following items:

- Registers
- Managing the Stack
- Function Call and Return
- Data Storage Formats
- Memory Section Usage
- Global Array Alignment
- Controlling System Heap Size and Placement
- Using Multiple Heaps
- Startup and Termination

Registers

The compiler makes use of the processor's registers in a variety of ways, as shown in the ***Processor Register Categorization*** table. Some registers fulfill more than role, depending on context.

This section contains:

- Dedicated Registers
- Preserved Registers
- Scratch Registers
- Stack Registers
- Parameter Registers
- Return Registers
- Aggregate Return Register
- Reservable Registers
- Alternate Registers

Table 2-46: Processor Register Categorization

Register	*Categorization*
R0	Scratch Register, Result Register
R1	Scratch Register, Long Result Register, Aggregate Result Register
R2	Scratch Register
R3	Preserved Register
R4	Scratch Register, Parameter Register
R5, R6, R7	Preserved Registers
R8	Scratch Register, Parameter Register
R9, R10, R11	Preserved Registers
R12	Scratch Register, Parameter Register
R13, R14, R15	Preserved Registers
S0-S15	Scratch Registers
I0, I1, I2, I3	Preserved Registers
I4	Scratch Register
I5	Preserved Register
I6	Frame Pointer
I7	Stack Pointer
I8, I9, I10, I11	Preserved Registers
I12	Return Address Register, Scratch Register
I13	Scratch Register
I14, I15	Preserved Registers
B0, B1, B2, B3	Preserved Registers
B4	Scratch Register
B5	Preserved Register
B6, B7	Stack Base Register
B8, B9, B10, B11	Preserved Registers
B12, B13	Scratch Registers
B14, B15	Preserved Registers
L0-L5	Preserved Registers, 0
L6, L7	Stack Length Register
L8-L15	Preserved Registers, 0
M0, M1, M2, M3	Preserved Registers

Table 2-46: Processor Register Categorization (Continued)

Register	*Categorization*
M4	Scratch Register
M5	Dedicated Register, 0
M6	Dedicated Register, 1
M7	Dedicated Register, -1
M8, M9, M10, M11	Preserved Registers
M12	Scratch Register
M13	Dedicated Register, 0
M14	Dedicated Register, 1
M15	Dedicated Register, -1
USTAT1-USTAT4	User Registers
MRF, MRB	When the -char-size-8 switch is enabled on processors with support for byte-addressing (see the ***Processors Supporting Byte-Addressing*** table in Processor Features), these are Scratch Registers. Otherwise they are Preserved Registers
MSF, MSB	Scratch Registers
ASTAT, ASTATy	Scratch Register
STKY, STKYy	Scratch Register
MODE1, MODE2	Preserved Registers
MMASK	Dedicated Register, 0xE03003

Dedicated Registers

The C/C++ run-time environment specifies a set of registers whose contents should not be changed except in specific defined circumstances. If these registers are changed, their values must be saved and restored. The dedicated register values must always be valid:

- On entry to any compiled function.
- On return to any compiled function.
- On exit from `asm` statements and interrupt handlers.

The dedicated registers are `I6`, `I7`, `B6`, `B7`, `M5-M7`, `M13-M15`, `MODE1`, `MODE2`, `MMASK`, and all `L` registers.

- `I7` and `I6` are the stack pointer and the frame pointer registers, respectively. `B7` and `B6` indicate the base of the stack, and `L7` and `L6` indicate the stack length.
- The remaining `L` registers define the lengths of the DAG's circular buffers. The compiler uses the DAG registers, both in linear mode and in circular buffering mode. The compiler assumes that the `Length` registers are zero, both on entry to functions and on return from functions, and ensures this is the case when it generates calls or returns. Your application may modify the Length registers and use the circular buffers, but you must

ensure that the Length registers are appropriately reset when calling compiled functions, or returning to compiled functions. Interrupt handlers must save and restore the Length registers, if using DAG registers.

- Registers M5–M7 and M13–M15 contain the fixed values of 0, 1 and -1 (M5 and M13 contain 0, M6 and M14 contain 1 and M7 and M15 contain -1).
- MODE1 and MODE2 are set to control the overall state of the processor, such as saturation, truncation and circular buffering. The compiler may change specific bits in these registers temporarily, in order to perform operations with specific semantics. For more information, see Mode Registers.
- The MMASK(Mode Mask) 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 by default: BR0, BR8, IRPTEN, ALUSAT, TRUNCATE, PEYEN, BDCST1, BDCST9.

  MMASK:RND32 can additionally be set if it is required that interrupt handlers are run with MODE1:RND32 cleared for 40-bit extended precision floats.

Mode Registers

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) assembly directives
- Disables ALU saturation (MODE1 register, ALUSAT bit = 0)
- Uses default FIX instruction rounding to nearest (MODE1 register, TRUNCATE=0)
- Enables circular buffering by setting CBUFEN on MODE1

Preserved Registers

These registers are also known as *callee-preserved registers*, as it is the callee's responsibility to ensure that these registers have the same value upon function return as they did upon entry to the function, regardless of whether the registers changed value in the meantime.

The C/C++ run-time environment specifies a set of registers whose contents must be saved and restored. Your assembly function must save these registers during the function's prologue and restore the registers as part of the function's epilogue. The call-preserved registers must be saved and restored if they are modified within the assembly function; if a function does not change a particular register, it does not need to save and restore the register.

The ***Callee-Preserved Registers*** table lists the default set of preserved registers.

Table 2-47: Callee-Preserved Registers

B0 *1	B1	B2	B3	B5	B8

Table 2-47: Callee-Preserved Registers (Continued)

B9	B10	B11	B14	B15	
I0	I1	I2	I3	I5	I8
I9	I10	I11	I14	I15	MODE1
MODE2	MRB [*2]	MRF [2]	M0	M1	M2
M3	M8	M9	M10	M11	R3
R5	R6	R7	R9	R10	R11
R13	R14	R15			

*1 If you use a callee-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.

*2 These registers are scratch registers when compiling with the `-char-size-8` switch on processors that support byte-addressing (see the ***Processors Supporting Byte-Addressing*** table in Processor Features).

NOTE: Functions may declare a non-standard partitioning of preserved/scratch registers through mechanisms such as #pragma regs_clobbered *string*, which any calling function must respect.

Scratch Registers

Scratch registers are also known as *caller-preserved registers*, as it is the caller's responsibility to ensure that the value of these registers is preserved across function calls, if required.

The C/C++ run-time environment specifies a set of fregisters whose contents need not be saved and restored. Note that the contents of these registers are not preserved across function calls.

The ***Scratch Registers*** table lists the default set of scratch registers.

Table 2-48: Scratch Registers

B4	B12	B13	R0	R1	R2	R4	R8	R12
I4	I12	I13	M4	M12	PX	ASTATy	STKYy	
S0	S1	S2	S3	S4	S5	S6	S7	S8
S9	S10	S11	S12	S13	S14	S15	MSB	MSF
MRB [*1]	MRF [1]							

*1 These registers are only scratch registers when compiling with the `-char-size-8` switch on processors that support byte-addressing (see the ***Processors Supporting Byte-Addressing*** table in Processor Features).

NOTE: Functions may declare a non-standard partitioning of preserved/scratch registers through mechanisms such as #pragma regs_clobbered *string*, which any calling function must respect.

Stack Registers

The C/C++ run-time environment reserves a set of registers for controlling the run-time stack. These registers may be modified for stack management in assembly functions, but they must be saved and restored. Never modify the stack registers within compiled functions. The ***Stack Registers*** table lists these registers.

Table 2-49: Stack 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.
`B6, B7`	Stack base address	Do not modify.
`L6, L7`	Stack length	Do not modify.

NOTE: The run-time environment makes use of the processor's circular-buffer mechanism for stack overflow detection: if the stack pointer (`I7`) advances beyond the limits defined by its corresponding base (`B7`) and length (`L7`) registers, a circular-buffer interrupt occurs (`CB7I`). Thus, it is essential that the `B7` and `L7` registers are set correctly.

Parameter Registers

When calling a function, the first three words of parameter data are passed to the callee in registers `R4`, `R8`, and `R12`.

Return Registers

When a function returns a value back to its caller, if the returned value is 64 bits or smaller in size, the value is returned in the `R0` register and, if necessary, in the `R1` register. In the case of 64-bit return values, when the char-size is set to 8 bits, `R0` contains the least significant word and `R1` contains the most significant word, else `R0` contains the most significant word and `R1` contains the least significant word.

Aggregate Return Register

When a function returns a value back to its caller, if the returned value is larger than 64 bits in size, the value is returned in space reserved on the stack. This stack space is allocated by the caller, and a pointer to the start of the space is passed to the callee by the caller in the `R1` register.

Reservable Registers

The `-reserve` *`register[, register …]`* 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. The ***Reservable Registers*** table lists the registers that can be reserved with this switch. Note that the C run-time library does not use these registers, unless explicitly noted in the function's description in the *C/C++ Library Manual for SHARC Processors*.

NOTE: Reserving registers can negatively influence the efficiency of compiled C/C++ code; use this option sparingly.

Registers in the USTAT class are never used by the compiler, except when explicitly directed via asm statements. Although USTAT registers may be included in clobber sets and specified by the -reserve switch, the compiler will not generate code to save or restore them.

Table 2-50: Reservable 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

Alternate Registers

With the exception of the background multiplier register MRB, which is visible at the same time as the foreground register MRF, the C/C++ run-time environment model does not use - or have any knowledge of - the alternate registers. To use these registers, you must first understand several aspects of the C/C++ run-time model.

The C/C++ run-time model uses register I6 as the frame pointer and register I7 as the stack pointer. Setting the DAG register set that contains I6 and I7 from a background register set to an active register set will have a direct affect on any stack operations, so you must ensure that I6 and I7 are set correctly before any stack operations are performed:

- Before switching to the alternate register set, save I6, B6 and L6, and I7, B7 and L7.
- After switching, restore I6, B6 and L6, and I7, B7 and L7 to the saved values.

If the background I6 and I7 registers are active and an interrupt occurs, the C/C++ run-time model will use I6 and I7 to update the stack. This will result in faulty stack handling if they are not set properly.

NOTE: If you do not intend to set I6 and I7, ensure that interrupts are disabled.

Managing the Stack

The C/C++ run-time environment uses the run-time stack to store automatic variables and return addresses. The stack is managed by a frame pointer (I6) and a stack pointer (I7) and grows downward in memory, moving from higher to lower addresses.

The stack pointer points to the location where the next value to be pushed will be stored, i.e. push operations write a value, then move the stack pointer to the next free address.

Whenever storing data on the stack, you must always decrement the stack pointer first, so that any data on the stack has an address that is higher than the current stack pointer value. Otherwise, data may be corrupted by interrupt handlers as they will save and restore context onto the top of the stack.

A *stack frame* is a section of the stack used to hold information about the current context of the C/C++ program. Information in the frame includes local variables, compiler temporaries, and parameters for the next function.

The frame pointer serves as a base for accessing memory in the stack frame. Routines refer to locals, temporaries, and parameters by their offset from the frame pointer.

The ***Example Run-Time Stack*** figure 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()`.

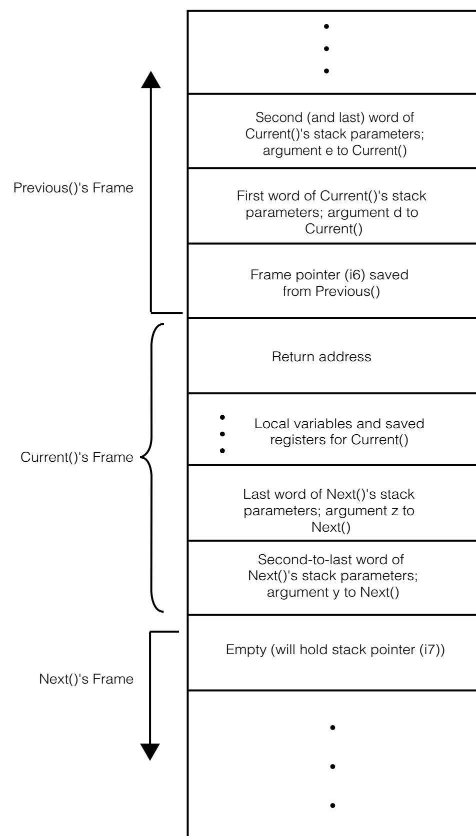

Figure 2-2: Example Run-Time Stack

NOTE: Stack usage for passing any or all of a function's arguments depends on the number and types of parameters to the function.

The prototypes for the functions in the ***Example Run-Time Stack*** figure 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 frame pointer (the I6 register) equal to the stack pointer (in the I7 register)
- Uses a delayed-branch instruction to pass control to the called function
- Pushes the old frame pointer (held in the R2 register) onto the run-time stack during the first branch delay slot
- Pushes the return address minus 1 (PC) onto the run-time stack during the second delayed-branch slot

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;
```

NOTE: When generating short-word code for processors that support VISA execution, .label-1 is used instead of 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 C/C++/Assembly Support Macros.

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 a 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 the current frame pointer (the i6 register), during the first branch delay slot
- Restores the caller's frame pointer, i6 register, by popping the previously saved frame pointer off the run-time stack and loading the value into i6 during the second delay-branch slot

The following instructions return from the function and restore the stack and frame pointers. Note that the restoring of the frame and stack pointers is 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 C/C++/Assembly Support Macros.

In the following code examples (***Stack Management Example, C Code***, ***Stack Management, Example ADSP-2116x Assembly Code*** and ***Stack Management, Example ADSP-SC589 Assembly Code compiled with -char-size-8***), 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.

NOTE: ***Stack Management***, ***Example ADSP-2116x Assembly Code and Stack Management***, ***Example ADSP-SC589 Assembly Code compiled with -char-size-8*** show non-optimized assembly code, for clarity; in optimized assembly code, the function prologue and epilogue instructions are often inter-mixed with instructions from the body of the function, for performance reasons.

Example 1. Stack Management, Example C Code

```
/* Stack management - C code */

int my_func(int, int);
int arg_a, return_c;

void foo(void)
{
int 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;
}
```

Example 2. Stack Management, Example ADSP-2116x Assembly Code

```
/* Stack management - C code compiled for ADSP-21160 */
   .section/pm/DOUBLE32 seg_pmco;
.epctext:
_my_func:
  /* Reserve space for local copies of parameters */
     modify(i7,-2);

  /* Save incoming parameters on stack */
     dm(-2,i6)=r8;
     dm(-3,i6)=r4;

  /* Compute result = (arg_1+arg_2)/2 */
  /* i.e. r0 = (r4+r8)/2 */
     r8=r4+r8;
     r12=lshift r8 by -31;
     r2=r8+r12;
     r0=ashift r2 by -1;

  /* Function return sequence */
```

```
     i12=dm(m7,i6);
     jump (m14,i12) (db); rframe; nop;
._my_func.end:
     .global _my_func;
     .type _my_func,STT_FUNC;

_foo:
  /* Reserve stack space for local variable arg_b */
     modify(i7,-2);

  /* Initialise arg_b to zero */
     dm(-2,i6)=m13;

  /* set first param register, r4, to arg_a */
     r4=dm(_arg_a);

  /* set second param register, r8, to arg_b, i.e. zero */
     r8=m5;

  /* Function call sequence */
     cjump _my_func (db); dm(i7,m7)=r2; dm(i7,m7)=pc;

  /* Store function result register, r0, into return_c */
     (_return_c)=r0;

  /* Function return sequence */
     i12=dm(m7,i6);
     jump (m14,i12) (db); rframe; nop;
._foo.end:
     .global _foo;
     .type _foo,STT_FUNC;
```

Stack Management, Example ADSP-SC589 Assembly Code compiled with -char-size-8 shows the same example compiled for ADSP-21589 with the `-char-size-8` switch. The code shows the differences in the symbol name mangling scheme used (a dot is append to the C identifier, rather than an underscore prefix), the use of VISA execution, and the use of the `(nw)` qualifier on the `modify` instruction to specify that the offset is in words.

Example 3. Stack Management, Example ADSP-SC589 Assembly Code compiled with -char-size-8

```
/* Stack management - C code compiled for ADSP-SC589 */
   .section/sw/DOUBLE32 seg_swco;
.epctext:
my_func.:
  /* Reserve space for local copies of parameters */
     modify(i7,-2) (nw);

  /* Save incoming parameters on stack */
     dm(-2,i6)=r8;
     dm(-3,i6)=r4;
```

```
  /* Compute result = (arg_1+arg_2)/2 */
  /* i.e. r0 = (r4+r8)/2 */
     r8=r4+r8;
     r12=lshift r8 by -31;
        r2=r8+r12;
        r0=ashift r2 by -1;
     /* Function return sequence */
        i12=dm(m7,i6);
        jump (m14,i12) (db); rframe; nop;
.my_func..end:
        .global my_func.;
        .type my_func.,STT_FUNC;
foo.:
     /* Reserve stack space for local variable arg_b */
        modify(i7,-2) (nw);
     /* Initialise arg_b to zero */
        dm(-2,i6)=m13;
     /* Set first param register, r4, to arg_a */
        r4=dm(arg_a.);
     /* set second param register, r8, to arg_b, i.e. zero */
        r8=m5;
     /* Function call sequence */
        cjump my_func. (db); dm(i7,m7)=r2; dm(i7,m7)=.LCJ0-1;
.LCJ0:
     /* Store function result register, r0, into return_c */
        dm(return_c.)=r0;
     /* Function return sequence */
        i12=dm(m7,i6);
        jump (m14,i12) (db); rframe; nop;
.foo..end:
        .global foo.;
        .type foo.,STT_FUNC;
```

When writing code for processors that support byte-addressing, care must be taken to ensure that the frame and stack pointers `i6` and `i7`, and their corresponding base registers, `b6` and `b7`, are in the correct address space on entry to the called function. If calling a byte-addressed function from byte-addressed code, or a word-addressed function from word-addressed code, no address space translation will be necessary. However if calling a word-addressed function from byte-addressed code, or vice versa, the frame and stack registers must be converted to the opposite address space before the call, and converted back again after returning to the callee. This can be achieved by using the `b2w` and `w2b` instructions to convert the `i7`, `b6` and `b7` registers (it is not necessary to explicitly convert `i6` since it is set from the `i7` register during the call sequence). For example, the following sequence may be used to call a byte-addressed function from word-addressed code:

```
i7=w2b(i7); // convert stack pointer
b7=w2b(b7); // convert stack base
b6=w2b(b6); // convert frame base
cjump f. (db); // call byte-addressed function
  dm(i7,m7)=r2;
  dm(i7,m7)=.LCJ0-1;
```

```
.LCJ0:
b6=b2w(b6); // convert frame base
b7=b2w(b7); // convert stack base
i7=b2w(i7); // convert stack pointer
```

The next two sections, Transferring Function Arguments and Return Value and Using Macros to Manage the Stack, provide additional detail on function call requirements.

Function Call and Return

The transfer of control from a calling function to a called function, and returning control back again, is the joint responsibility of the calling function and the called function. The calling function has to pass the appropriate parameters, in registers or upon the stack, and in some cases has to provide space for the return value too. The called function has to keep its own temporary workspace separate from that of its caller. Both are required to ensure the integrity of some parts of the register set.

From the caller's point of view, the sequence of actions looks like this:

- If the function being called clobbers registers that contain values being used by the caller, the caller must save those values on the stack prior to making the call. In the ***Example Run-Time Stack*** figure (see Managing the Stack), this is the "saved registers".
- If the called function returns an aggregate value that is returned via the stack, the caller must allocate stack space for this returned value. See Aggregate Return Register.
- If the called function takes parameters, the caller must set up those parameters, either in registers or on the stack. In the ***Example Run-Time Stack*** figure, this is "Next()'s stack parameters".
- The caller can now call the function.
- After the function returns, the caller must reset the stack pointer, to dispose of "Next()'s stack parameters", and restore any needed registers that might have been clobbered by the called function.

From the callee's point of view, the sequence of actions looks like this:

- Upon entry to the callee, the stack pointer will point to the top of the "First word of Current()'s stack parameters" area of the ***Example Run-Time Stack*** figure. Note that this figure is viewed differently by caller and callee: "Next()'s stack parameters" of the caller are "Current()'s stack parameters" as far as the callee is concerned.
- If the function will be calling any further functions, it will have to save the Frame Pointer, `I6`.
- If the function needs any space for temporaries, it must create the "local variables" space on the stack.
- If the function needs to modify any registers that are not considered scratch registers, the function must save their current values prior to changing them. In other words, the function must preserve the value of any callee-preserved registers.
- The function may now perform its main task.
- Upon completion, the function may need to return a value to the caller. To do this, it must either load the value into the Result Registers, or store it to the stack.

- Prior to returning, the function must restore the value of any callee-preserved registers it has modified.
- The function must pop the "local variables" space from the stack, restore the `RETS` value, restore the caller's Frame Pointer value (if changed) and restore the Stack Pointer to the value it had on entry to the function. These operations are all combined; restoring the stack pointer to its previous position discards the "local variables" space.
- The function must retrieve the return address from the stack, where it was placed by the caller.
- Finally, the function can return control back to the caller.

Further details are given in the following sections:

- Transferring Function Arguments and Return Value
- Calling Assembly Subroutines From C/C++ Programs
- Calling C/C++ Functions From Assembly Programs
- C/C++/Assembly Support Macros
- Symbol Names in C/C++ and Assembly
- Implementing C++ Member Functions in Assembly Language
- Writing C/C++-Callable SIMD Subroutines
- Mixed C/C++/Assembly Programming Examples
- Exceptions Tables in Assembly Routines

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/C++ functions.

This section covers:

- Basic Argument Passing
- Passing Parameters for Variable Argument Lists
- Passing a C++ Class Instance
- Return Values
- Parameter and Return Value Examples

Basic Argument Passing

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. When the `-char-size-8` switch is used on processors that support byte-addressing, any parameters whose size is less than 32 bits are sign- or zero-extended to 32 bits by the caller.

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 than a single 32-bit word, then that parameter and all subsequent parameters are passed on the stack.
- Functions with variable-length parameter lists. For more information, see Passing Parameters for Variable Argument Lists.

The ***Parameter and Return Value Transfer Registers*** table 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 2-51: Parameter and Return Value Transfer Registers

Register	*Parameter Type Passed Or Returned*
R4	Pass first parameter of 32 bits or less.
R8	Pass second parameter of 32 bits or less.
R12	Pass third parameter of 32 bits or less.
stack	Pass fourth and remaining parameters; see exceptions to this rule on this page.
R0	Return int, long, char, float, short, pointer, and one-word structure parameters.
R0, R1	Return long double, long long, unsigned long long and two-word structure parameters. When using char-size 8, place MSW in R1 and LSW in R0, else 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.

Passing Parameters for Variable Argument Lists

The details of argument passing change slightly for variable argument lists.

For example, a function declared as follows may receive two or more arguments.

```
int varying(char *fmt, int flag, ...) { /* ... */ }
```

Variable argument lists are processed using the macros defined in the stdarg.h header file. The va_start() function obtains a pointer to the list of arguments which may be passed to other functions, or which may be walked by the va_arg() macro.

To support this, the compiler passes the last named argument - flag, in this case - on the stack, even if it would normally have been passed in R4, R8 or R12 (in this case, it would have been passed in R8). Any following arguments after the last named argument are also passed on the stack.

The va_start() function can then take the address of the last non-varying argument (flag, in the example above), and va_arg() can walk through the complete argument list on the stack.

Any arguments before the last named parameter are passed as normal. In this case, the first argument, fmt, is passed in R4.

Passing a C++ Class Instance

A C++ class instance function parameter is always passed by reference when a copy constructor has been defined for the C++ class. If a copy constructor has not been defined for the C++ class then the C++ class instance function parameter is passed by value.

Consider the following example:

```
class fr
{
    public:
      int v;
      fr () {}
      fr (const fr& rc1) : v(rc1.v) {}
};

extern int fn(fr x);

fr Y;

int main() {
  return fn (Y);
}
```

The function call `fn (Y)` in `main` will pass the C++ class instance `Y` by reference because a copy constructor for that C++ class has been defined by `fr (const fr& rc1) : v(rc1.v) {}`. If this copy constructor were removed, then `Y` would be passed by value.

Return Values

Values are usually returned from a called function to the caller in register `R0`, or in the register pair `R0-R1`, if necessary. The details are as follows:

- Arithmetic values of size 32 bits or less are returned in `R0`. Values of less than 32 bits in size (which are only used when compiling with the `-char-size-8` switch on processors that support byte-addressing) are sign- or zero-extended to 32 bits by the called function.
- 64-bit arithmetic types are returned in `R0` and `R1`. When the char-size is set to 8 bits, the least significant bits are held in `R0`, else they are held in `R1`.
- Pointer values are returned in `R0`.
- Aggregate types of 32 bits or less are returned in `R0`.
- Aggregate types of 64 bits in size are returned in `R0` and `R1`, with the lower-addressed bytes in `R0`.
- Aggregate values larger than 64 bits in size are returned on the stack. The caller must allocate sufficient space on the stack within the caller's own frame, and load the address of the lowest-addressed part of this storage into register `R1` before calling the function.

Parameter and Return Value Examples

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. If the -char-size-8 switch is being used on processors that support byte-addressing (see the ***Processors Supporting Byte-Addressing*** table in Processor Features), then c is sign-extended to 32 bits before being stored in register R12. The fourth argument, d, is passed on the stack.

This next example illustrates the effects of passing long double arguments.

```
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 each of these parameters fits in a single-word. 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. For processors that do not support byte-addressing, or when the -char-size-32 switch is used, it is passed in big-endian format. In this case, the most significant word is at I6 + 3 and the least significant word is at I6 + 4. If the -char-size-8 switch is being used, the value is passed in little-endian format. In this case, the least significant word is at I6 + 3 and the most significant word is at I6 + 4. For more information on the endianness of 64-bit types, see Endianness.

Calling Assembly Subroutines From C/C++ Programs

Before calling an assembly language subroutine from a C/C++ program, create a prototype to define the arguments for the assembly language subroutine and the interface from the C/C++ program to the assembly language subroutine. Even though it is legal to use a function without a prototype in C/C++, prototypes are a strongly-recommended practice for good software engineering. When the prototype is omitted, the compiler cannot perform argument-type checking and assumes that the return value is of type integer and uses K&R promotion rules instead of ANSI promotion rules.

C, C++ and assembly code use different namespaces for symbols. Refer to Symbol Names in C/C++ and Assembly for ways to specify an assembly routine from C/C++.

The compiler will assume that the called assembly function will obey the run-time model's rules on register usage. Refer to Registers for details.

NOTE: Functions may declare a non-standard partitioning of preserved/scratch registers through mechanisms such as #pragma regs_clobbered *`string`*, which any calling function must respect. If the assembly function being called from C/C++ uses a non-standard clobber set, declare this in the prototype.

The compiler also assumes that the machine state does not change during execution of the assembly language subroutine. If you change modes within your assembly routine-for example, enabling saturation via the `MODE` registers-ensure that you restore them to their previous value before returning.

Calling C/C++ Functions From Assembly Programs

C/C++ functions can be called from assembly code. The situation is similar to that described in Calling Assembly Subroutines From C/C++ Programs:

- The namespaces for C/C++ and assembly code are different; refer to Symbol Names in C/C++ and Assembly for details on how to specify a C/C++ function that can be referenced from assembly.
- The C/C++ function will obey the run-time model's rules described in Registers, so your calling assembly code must respect this, by not expecting caller-preserved registers to maintain their values over the call.
- If your assembly code is passing parameters to the C/C++ function or receiving a return value from it, you must follow the rules described in Transferring Function Arguments and Return Value.

There are additional requirements you must fulfil when calling C/C++ code from assembly code, however:

- You must ensure that the system stack is valid, as described in Managing the Stack.
- You must ensure that Dedicated Registers have their correct values.
- You must ensure that a system heap is set up. This is done for you if you are using the default or generated startup code and `.ldf` files. For more information, see Startup and Termination.

C/C++/Assembly Support Macros

This section describes macros defined in the `asm_sprt.h` system header file. Use these macros in your assembly language modules, to simplify interfacing with C/C++ functions.

NOTE: 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.

The following macros are available.

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.

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 off the stack and puts the value in the indicated register.

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 reads a value off 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` removes 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.

Symbol Names in C/C++ and Assembly

You can use C/C++ symbols (function or variable names) in assembly routines and use assembly symbols in C/C++ code. This section describes how to name and use C/C++ and assembly symbols.

Only global C/C++ symbols can be referenced from assembly source.

To use a C/C++ function or variable in an assembly routine, declare it as global in the C program. Import the symbol into the assembly routine by declaring the symbol with the `.EXTERN` assembler directive.

To use an assembly function or variable in your C/C++ program, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern` in the C program.

The ***C/C++ Naming Conventions for Symbols*** table shows several examples of the C/C++ and assembly interface naming conventions.

Table 2-52: Parameter and Return Value Transfer Registers

In the C/C++ Program	*In the Assembly Subroutine*

`int c_var; /*declared global*/`	`.extern _c_var;`
	`.type _c_var,STT_OBJECT;`
	or
	`.extern c_var.;`
	`.type c_var.,STT_OBJECT;`

`void c_func(void);`	`.global _c_func;`
	`.type _c_func,STT_FUNC;`
	or
	`.global c_func.;`
	`.type c_func.,STT_FUNC;`

`extern int asm_var;`	`.global _asm_var;`
	`.type _asm_var,STT_OBJECT;`
	`.byte = 0x00,0x00,0x00,0x00;`
	or

Table 2-52: Parameter and Return Value Transfer Registers (Continued)

In the C/C++ Program	*In the Assembly Subroutine*

	`.global asm_var.;`
	`.type asm_var.,STT_OBJECT;`
	`.byte = 0x00,0x00,0x00,0x00;`

`extern void asm_func(void);`	`.global _asm_func;`
	`.type _asm_func,STT_FUNC;`
	`_asm_func:`
	or
	`.global asm_func.;`
	`.type asm_func.,STT_FUNC;`
	`asm_func.:`

The following sections cover different approaches:

- C/C++ and Assembly: Extern Linkage
- C and Assembly: Underscore Prefix or Dot Suffix
- Other Approaches

C/C++ and Assembly: Extern Linkage

The compiler supports the use of extern to declare symbol names in the different C, C++ and assembly namespaces. For example:

```
extern int def_fn(void);
                     // "_def_fn", "def_fn.", "__Z6def_fnv" or "_Z6def_fnv."
extern "asm" int asm_fn(void);
                    // "asm_fn" in assembly
extern "C" int c_fn(void);
                    // "_c_fn" or "c_fn." in assembly
```

When compiling your source in C or C++ mode, you can use extern `"asm"` or `extern "C"` to specify which namespace you want your external symbols to use. Without the external linkage specifier, your symbol will use C namespace when compiling in C mode, and C++ namespace (mangled) when compiling in C++ mode.

C and Assembly: Underscore Prefix or Dot Suffix

As can be seen in C/C++ and Assembly: Extern Linkage, when the compiler generates the assembly version of a C-namespace symbol, it either prepends an underscore or appends a dot to the symbol name. When compiling with the `-char-size-8` switch on processors that support byte-addressing, the symbol name will have a dot suffix, otherwise it will have an underscore prefix. You can take advantage of this in your assembly source when referring to C-mode symbols, by adding the underscore or dot yourself.

Other Approaches

In addition to the external linkage feature described in C/C++ and Assembly: Extern Linkage, you can also use the following approaches in your C/C++ source:

- When declaring functions, you can provide an alternative linkage name, using #pragma linkage_name *identifier* or #pragma function_name *identifier*.
- When declaring variables in C, you can provide an alternative linkage name, using Keyword for Specifying Names in Generated Assembler (asm).
- When declaring functions in C, you can use Function, Variable and Type Attribute Keyword (__attribute__) to specify aliases of functions.

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 Symbol Names in C/C++ and Assembly, 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(void)
{
```

```
      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:
      r0=r4+r4;
      i12=dm(m7,i6);
      jump(m14,i12) (db);
      rframe;
      nop;
```

Note that the above example is for processors that do not support byte-addressing, or when using the `-char-size-32` switch. When using the `-char-size-8` switch, the function `cc_array` should appear in the assembly file as `cc_array`.

Writing C/C++-Callable SIMD Subroutines

You can write assembly subroutines that use SIMD mode 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.

NOTE: While it is possible to write subroutines that can be called in SIMD mode (i.e., the processor is in SIMD mode before the call and after the return), the compiler does not support a SIMD call interface. 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. For more information, see SIMD Support.

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 manual for the appropriate processor.

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

- Using Macros to Manage the Stack
- Stack Alignment
- Using Scratch Registers
- Using Void Functions
- Using the Stack for Arguments
- Using Registers for Arguments and Return
- Using Non-Leaf Routines That Make Calls
- Using Call Preserved Registers

Leaf routines are routines that return without making any calls. *Non-leaf routines* call other routines before returning to the caller.

You should use `cc21k` to compile your C/C++ program and assemble your assembly language modules. `cc21k` will invoke the compiler for C and C++ files, and will invoke the assembler for assembly files, defining common sets of macros, passing processor-specific switches and so on.

For example, the following `cc21k` command line

```
cc21k my_prog.c my_sub1.asm
```

runs `cc21k` with the following modules listed in the ***Modules for Running cc21k*** table. For more information, see Compiler Components.

Table 2-53: Modules for Running cc21k

Module	*Description*
`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

Using Inline Assembly

The following example shows how to write a simple routine in ADSP-21xxx/SCxxx assembly code that properly interfaces to the C/C++ environment. It uses the `asm()` construct to pass inline assembly code to the compiler.

```
int add(int x, int y, int z)
{
   int res;
   asm("%0=%2+%1; %0=%0+%3;":
        "=d"(res):"d"(x),"d"(y),"d"(z):"astat");
   return res;
}
```

For more information, see Keyword for Specifying Names in Generated Assembler (asm).

Using Macros to Manage the Stack

The ***Subroutine Return Address Example - C Code*** and ***Subroutine Return Address Example - Assembly Code*** listings 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 C/C++/Assembly Support Macros.

Example 1. 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;

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

Example 2. 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  */
_asm_func.end:
```

The above example shows code that works with processors that do not support byte-addressing, or when using the `-char-size-32` switch. When the `-char-size-8` switch is being used on processors that support byte-addressing, symbol names will be mangled according to the rules in C and Assembly: Underscore Prefix or Dot Suffix.

Stack Alignment

By default the compiler will always create stack frame sizes that are an even numbers of words, so that the stack pointer preserves its alignment if it is double-word aligned. However calling a function from an interrupt handler or assembly routine that does not preserve stack alignment will result in a misaligned stack.

As a result, the compiler does not attempt to take advantage of the double-word alignment of the stack unless you tell the compiler that the stack always remains double-word aligned by using additional alignment assertions in your source code. Therefore there is no requirement to maintain double-word alignment in assembly code routines unless you want to keep aligned data on the stack. If you align local variables using `#pragma align`, for example, you must keep the stack pointer aligned to a double-word boundary in assembly routines that call into C code.

Using Scratch Registers

To write assembly functions that can be called from a C/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. For more information, see Scratch Registers.

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 */
_dot.end:
```

The above example shows code that works with processors that do not support byte-addressing, or when using the –char-size-32 switch. When the -char-size-8 switch is being used on processors that support byte-addressing, symbol names will be mangled according to the rules in C and Assembly: Underscore Prefix or Dot Suffix.

Using Void Functions

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. For more information, see Registers.

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. A different mangling scheme is used when using the

`-char-size-8` switch on processors that support byte-addressing. For more information, see Symbol Names in C/C++ and Assembly.

```
/* 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 */
_delay.end:
```

Using the Stack for Arguments

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. For more information, see Function Call and Return and Symbol Names in C/C++ and Assembly.

```
/* 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;
_add5.end:
```

The above example shows code that works with processors that do not support byte-addressing, or when using the `-char-size-32` switch. When the `-char-size-8` switch is being used on processors that support byte-addressing, symbol names will be mangled according to the rules in C and Assembly: Underscore Prefix or Dot Suffix.

Using Registers for Arguments and Return

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`).

For more information, see Function Call and Return and Symbol Names in C/C++ and Assembly.

```
/* 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;
_add2.end:
```

Using Non-Leaf Routines That Make Calls

A more complicated example, which calls another routine, computes the length of the hypotenuse of a right-angled triangle:

$$z = \sqrt{x^2 + y^2}$$

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. For more information, see Function Call and Return. Function symbol names are discussed in Symbol Names in C/C++ and Assembly.

```
/* Non-Leaf Assembly Routines Example - _hypot */
/* float hypot(float x, float y);
 An assembly language subroutine to return z = (x^2 + y^2)^(1/2) */

#include <asm_sprt.h>

.section/pm seg_pmco;
.extern _sqrtf;
.global _hypot;
_hypot:
   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, _hypot returns the result to its caller in f0
   (and it is already there) */
   exit;               /* last line of non-leaf routine */
_hypot.end:
```

If a called function takes more than three single word parameters, the remaining parameters must be pushed onto 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. 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;
_calladd5.end:
```

Using Call Preserved Registers

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. For more information, see Preserved Registers. Function and data symbol names are discussed in Symbol Names in C/C++ and Assembly.

```
/* 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
*/

      r2=i6;
      i6=i7;
      jump (m13,i8) (db);
      dm(i7,m7)=r2;
      dm(i7,m7)=pc;

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;
_pass_array.end:
```

Exceptions Tables in Assembly Routines

C++ functions can throw C++ exceptions, which must be caught by another function earlier in the call-stack. Part of this catching process involves unwinding the stack of intervening, still-active function calls. The C++ exception support library uses additional function details to perform this unwinding. The exception support gets this information from different places:

- When C++ modules are compiled with exceptions enabled by the `-eh` switch, the compiler generates the necessary unwinding tables.
- When C modules are compiled, exceptions information is not usually necessary, but the compiler will generate unwinding information if the `-eh` switch is specified.
- Assembly modules are not compiled, so unwinding information must be supplied manually, if necessary.

Assembly functions rarely need to provide exception-unwinding information. It is only necessary when all of the following conditions apply:

- The assembly routine may be called by a C or C++ function.

- The assembly routine calls a C++ function (or a C function that may lead to a C++ function being called, while the assembly routine is still active).
- The called C++ function may throw an exception.

The assembly routine must allocate a stack frame as described in Managing the Stack. On entry to the assembly routine, call-preserved registers (see Preserved Registers) that are modified in the routine should be saved into a contiguous region within the stack frame, called the *save* area. Registers are saved at ascending addresses in the save area in the order given in the ***Function Exception Table Register Numbers*** table.

A word in the `.gdt` section must be initialized with the address of the function exceptions table. This word must be marked with the `.RETAIN_NAME` directive to prevent it being removed by linker data elimination. The function exceptions table itself must be initialized as illustrated in the ***Function Exceptions Table*** table.

Table 2-54: Function Exceptions Table

Offset (words)	*Size (words)*	*Description*
0	1	Start address of the routine
1	1	First address after end of routine
2	1	Signed offset from frame pointer of register save area
3	4	Bit set indicating which registers are saved
8	1	Always zero. Indicates this is not C++ code

The bit set field of the function exceptions table contains a bit for each register. The bits corresponding to registers saved in the save area must be set to one and the other bits set to zero. The bit numbers corresponding to each register are given in the ***Function Exception Table Register Numbers*** table, where bit 0 is the least significant bit of the lowest addressed word, bit 31 the most significant bit of that word, bit 32 the least significant bit of the second lowest addressed word and so on.

Bit numbering may best be explained by the C code to test bit number,

```
int wrd = r/32;
int bit = lu << (r%32);
if (bitset[wrd] & bit)
    /* register r was saved */
```

Table 2-55: Function Exception Table Register Numbers

Register	*Bit Number*	*Words Taken in Save Area if Saved*
ASTAT	0	1
ASTATY	1	1
R0 - R15	2 - 17	1
S0 - S15	18 -33	1
M0 - M15	34 - 49	1

Table 2-55: Function Exception Table Register Numbers (Continued)

Register	*Bit Number*	*Words Taken in Save Area if Saved*
B0 - B15	50 - 65	1
I0 - I15	66 - 81	1
L0 - L15	82 - 97	1
MRF	98	3
MSF	99	3
MRB	100	3
MSB	101	3
PX1, PX2	102 - 103	1
USTAT1 - USTAT4	104 - 107	1

This example shows an assembly routine with function exceptions table, showing how to write code for processors that do not support byte-addressing, or when using the `-char-size-32` switch.

```
.section/pm seg_pmco;
_asmfunc:
.LN._asmfunc:
   modify(i7,-6);          // allocate stack frame
                           // save area at I6-7
   dm(-7,i6)=r5;           // save_area[0] = r5
   dm(-6,i6)=r6;           // save_area[1] = r6
   dm(-5,i6)=r7;           // save_area[2] = r7
   r2=i0; dm(-4,i6)=r2;    // save_area[3] = i0
   r2=i1; dm(-3,i6)=r2;    // save_area[4] = i1
   r2=i2; dm(-2,i6)=r2;    // save_area[5] = i2
   // use R5,R6,R7,I0,I1,I2, call a C++ function
   i0=dm(-4,i6);
   i1=dm(-3,i6);
   i2=dm(-2,i6);
   r5=dm(-7,i6);
   r6=dm(-6,i6);
   r7=dm(-5,i6);
   i12=dm(m7,i6);
   jump (m14,i12) (db); rframe; nop;
.LN._asmfunc.end:
._asmfunc.end:
.global _asmfunc;
.type _asmfunc, STT_FUNC;
.section/dm .edt;   // conventionally function exceptions
                    // tables go in .edt
.var .function_exceptions_table[8] =
.LN._asmfunc,       // first address of _asmfunc
.LN._asmfunc.end,   // first address after _asmfunc
-7,                 // offset of save area from I6
```

```
0x00000380, 0, 0x0000001c, 0,
                    // bit set, bits 7=R5,8=R6,9=R7,66=I0,67=I1,68=I2
0;                  // always zero for non-c++
.section/dm .gdt;
.align 1;
.fet_index:
.var = .function_exceptions_table;
                    // address of table in .gdt
.retain_name .fet_index;
```

When using the `-char-size-8` switch on processors that support byte-addressing, the example should be written to use byte-addressed data sections and use symbol names suitable for use with byte-addressed code:

```
.section/pm seg_pmco;
asmfunc.:
.LN.asmfunc.:
   modify(i7,-6) (nw);    // allocate stack frame
                          // save area at I6-7
   dm(-7,i6)=r5;          // save_area[0] = r5
   dm(-6,i6)=r6;          // save_area[1] = r6
   dm(-5,i6)=r7;          // save_area[2] = r7
   r2=i0; dm(-4,i6)=r2;   // save_area[3] = i0
   r2=i1; dm(-3,i6)=r2;   // save_area[4] = i1
   r2=i2; dm(-2,i6)=r2;   // save_area[5] = i2
   // use R5,R6,R7,I0,I1,I2, call a C++ function
   i0=dm(-4,i6);
   i1=dm(-3,i6);
   i2=dm(-2,i6);
   r5=dm(-7,i6);
   r6=dm(-6,i6);
   r7=dm(-5,i6);
   i12=dm(m7,i6);
   jump (m14,i12) (db); rframe; nop;
.LN.asmfunc..end:
.asmfunc..end:
.global asmfunc.;
.type asmfunc., STT_FUNC;
.section .edt;      // conventionally function exceptions
                    // tables go in .edt
.var .function_exceptions_table[8] =
.LN._asmfunc,       // first address of _asmfunc
.LN._asmfunc.end,   // first address after _asmfunc
-7,                 // offset of save area from I6
0x00000380, 0, 0x0000001c, 0,
                    // bit set, bits 7=R5,8=R6,9=R7,66=I0,67=I1,68=I2
0;                  // always zero for non-c++
.section .gdt;
.align 4;
.fet_index:
.var = .function_exceptions_table;
```

```
                    // address of table in .gdt
.retain_name .fet_index;
```

Note that since the exceptions table is written for either `-char-size-8` or `-char-size-32` code, the function must be called from code compiled with the same char-size switch.

Data Storage Formats

This section explains how the compiler stores some kinds of data. It covers the following topics:

- Using Data Storage Formats
- Floating-Point Data Size
- Floating-Point Binary Formats
- fract Data Representation
- Precision Restrictions With 40-Bit Floating-Point Arithmetic

Using Data Storage Formats

The sizes of intrinsic C/C++ data types are selected by Analog Devices so that normal C/C++ programs execute with hardware-native data types, and, therefore, at high speed. The C/C++ run-time environment uses the intrinsic C/C++ data types and data formats that appear in the ***Data Storage Formats and Data Type Sizes*** table, the ***Data Storage Formats and Data Storage*** table, the ***Data Storage Format for Float and Double Types*** figure, and the ***Double-Precision IEEE Format*** figure.

NOTE: Except for the 64-bit floating-point types on processors with native double-precision floating-point support (see the ***Processors Support Native Double-Precision Floating-Point Arithmetic*** table in Processor Features), the 64-bit data types are implemented using software emulation, and are expected to run more slowly than hardware-supported native data types. The 64-bit data types are `long double`, `long long`, and `unsigned long long`. When the `-double-size[-32|-64]` switch is specified, double is also a 64-bit data type.

Table 2-56: Data Storage Formats and Data Storage

Data	*Storage Format*
`long long`	For processors without byte-addressing support, or when using the `-char-size-32` switch, writes 64-bit two's complement data with the most significant word closer to address `0x0`, proceeding toward the top of memory with the rest. When using the `-char-size-8` switch on processors with byte-addressing support, writes 64-bit two's complement data with the least significant byte closer to address `0x0`, proceeding toward the top of memory with the rest.

Table 2-56: Data Storage Formats and Data Storage (Continued)

Data	*Storage Format*
unsigned long long	For processors without byte-addressing support, or when using the -char-size-32 switch, writes 64-bit magnitude data with the most significant word closer to address 0x0, proceeding toward the top of memory with the rest. When using the -char-size-8 switch on processors with byte-addressing support, writes 64-bit magnitude data with the least significant byte closer to address 0x0, proceeding toward the top of memory with the rest.
long double	For processors without byte-addressing support, or when using the -char-size-32 switch, writes 64-bit IEEE double-precision data with the most significant word closer to address 0x0, proceeding toward the top of memory with the rest. When using the -char-size-8 switch on processors with byte-addressing support, writes 64-bit IEEE double-precision data with the least significant byte closer to address 0x0, proceeding toward the top of memory with the rest. See the ***Processors Supporting Native Double-Precision Floating-Point Arithmetic*** table in Processor Features for details.

Floating-Point Data Size

On SHARC processors, the float data type is 32 bits, and the double data type default size is 32 bits. This size is chosen because it is the most efficient. The 64-bit long double data type is available if more precision is needed, although this is more costly because on processors without native double-precision floating-point support, the type exceeds the data sizes supported natively by hardware.

In the C language, floating-point literal constants default to the double data type. When operations involve both float and double, the float operands are promoted to double and the operation is done at double size. By having double default to a 32-bit data type, the SHARC compiler usually avoids additional expense during these promotions. This does not, however, fully conform to the ISO/IEC 9899:1990 C standard, the ISO/IEC 9899:1999 C standard, and the ISO/IEC 14882:2003 C++ standard, all of which require that the double type supports at least 10 digits of precision.

The -double-size[-32|-64] switch sets the size of the double type to 64 bits if additional precision, or full standard conformance, is required.

The -double-size-64 switch causes the compiler to treat the double data type as a 64-bit data type, instead of a 32-bit data type. This means that all values are promoted to 64 bits, and consequently incur more storage and cycles during computation. The switch does not affect the size of the float data type, which remains at 32 bits.

Consider the following case.

```
float add_two(float x) { return x + 2.0; }   // has promotion
```

When compiling this function, the compiler promotes the float value x to double, to match the literal constant 2.0. The addition becomes a double operation, and the result is truncated back to a float before being returned.

By default, or with the -double-size[-32|-64] switch, the promotion and truncation operations are empty operations-they require no work because the float and double types default to the same size. Thus, there is no cost.

With the -double-size-64 switch, the promotion and truncation operations require work because the double constant 2.0 is a 64-bit value. The x value is promoted to 64 bits, a 64-bit addition is performed, and the result is truncated to 32 bits before being returned.

In contrast, since the literal constant 2.0f in the following example has an f suffix, it is a float-type constant, not a double-type constant.

```
float add_two(float x) { return x + 2.0f; }   // no promotion
```

Thus, both operands to the addition are of type float, and no promotion or truncation is necessary. This version of the function does not produce any performance degradation when the -double-size-64 switch is used.

You must be consistent in your use of the -double-size-{32|64} switch.

Consider the two files, such as:

```
file x.c:codeblock

double add_nums(double x, double y) { return x + y; }
file y.c:codeblock
codeblockextern double add_nums(double, double);codeblock
codeblockdouble times_two(double val) { return add_nums(val, val); }
```

Both files must be compiled with the same usage of -double-size{32|64}. Otherwise, times_two() and add_nums() will be exchanging data in mismatched formats, and incorrect behavior will occur. The ***Use of the -double-size-{32|64}*** switch table shows the results for the various permutations.

Table 2-57: Use of the -double-size-{32|64} switch

x.c	*y.c*	*Result*
default	default	Okay
default	-double-size-32	Okay
-double-size-32	default	Okay
-double-size-32	-double-size-32	Okay
-double-size-64	-double-size-64	Okay
-double-size-32	-double-size-64	Error
-double-size-64	-double-size-32	Error

If a file does not make use of any double-typed data, it may be compiled with the-double-size-any switch, to indicate this fact. Files compiled in this way may be linked with files compiled with -double-size-32 or with -double-size-64, without conflict.

Conflicts are detected by the linker and result in linker error li1151, *Input sections have inconsistent qualifiers*.

Floating-Point Binary Formats

This section covers:

- IEEE Floating-Point Format
- IEEE Floating-Point Implementation

IEEE Floating-Point Format

By default, the SHARC compiler provides floating-point arithmetic using IEEE single- and double-precision formats. Single-precision IEEE format (the ***Data Storage Format for Float and Double Types*** figure) provides a 32-bit value, with 23 bits for the mantissa, 8 bits for the exponent, and 1 bit for the sign. This format is used for the `float` data type, and for the `double` data type by default and when the `-double-size-32` switch is used. The 32-bit `double` data type violates the ISO/IEC 9899:1990 C standard, the ISO/IEC 9899:1999 C standard, and the ISO/IEC 14882:2003 C++ standard.

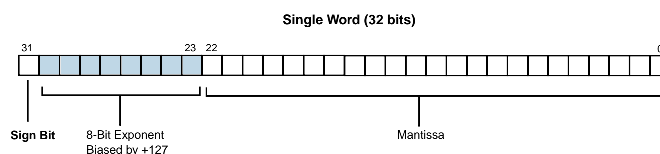

Figure 2-3: Data Storage Format for Float and Double Types

In the ***Data Storage Format for Float and Double Types*** figure, the single word (32-bit) data storage format equates to:

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

where

- Sign - comes from the sign bit.
- Mantissa - represents the fractional part of the mantissa 23 bits. (The "1." is assumed in this format.)
- Exponent - represents the 8-bit exponent.

Double-precision IEEE format (the ***Double-Precision IEEE Format*** figure) provides a 64-bit value, with 52 bits for the mantissa, 11 bits for the exponent, and 1 bit for the sign. This format is used for the `long double` data type, and for the `double` data type when the `-double-size-64` switch is used. A 64-bit value for the `double` data type is compliant to with the ISO/IEC 9899:1990 C standard, the ISO/IEC 9899:1999 C standard, and the ISO/IEC 14882:2003 C++ standard. See Language Standards Compliance.

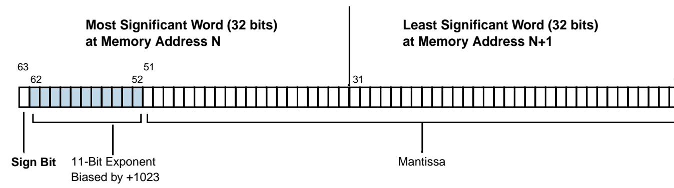

Figure 2-5: Double-Precision IEEE Format

In the ***Double-Precision IEEE Format*** figure, the two-word (64-bit) data storage format equates to:

$$-1^{Sign} \times 1.Mantiss\ a \times 2^{(Exponent - 1023)}$$

where

- Sign - comes from the sign bit.
- Mantissa - represents the fractional part of the mantissa 52 bits. (The "1." is assumed in this format.)
- Exponent - represents the 11-bit exponent.

IEEE Floating-Point Implementation

The following points should be noted about IEEE floating-point support for SHARC:

- For 32-bit (single-precision) arithmetic, the majority of operations are implemented using the processor's native hardware.
- For processors without native double-precision floating-point support (see the ***Double-Precision IEEE Format*** figure in IEEE Floating-Point Format), 64-bit (double-precision) arithmetic operations are emulated using a software library.
- All NaNs are handled as quiet NaNs. Signalling NaNs are not supported.
- The sign bit of the NaN is not guaranteed to be preserved by library routines and type conversions.
- In general, denormalized numbers are flushed to zero before being used in arithmetic.

fract Data Representation

The `fract` types are native fixed-point types that can be used to write code using saturating, fixed-point arithmetic. The native fixed-point types are discussed in Using Native Fixed-Point Types.

The `short fract`, `fract` and `long fract` type represent a 32-bit signed fractional value. All types have the range [-1.0,+1.0).

The `short fract`, `fract`, and `long fract` data representations are shown in the following figure.

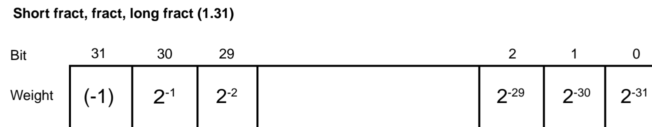

Figure 2-5: Data Storage Format for short fract, fract, and long fract (1.31)

Therefore, to represent 0.25 in `fract`, the HEX representation is `0x20000000` (2^{-2}). For -1, the hexadecimal representation in `fract` is `0x80000000`. `short fract`, `fract`, and `long fract` cannot represent +1 exactly, but they get quite close with `0x7fffffff`.

The `unsigned short fract`, `unsigned fract`, and `unsigned long fract` types represent a 32-bit unsigned fractional value. All types have the range [0.0,+1.0).

The `unsigned short fract`, `unsigned fract`, and `unsigned long fract` data representations are shown in the following figure.

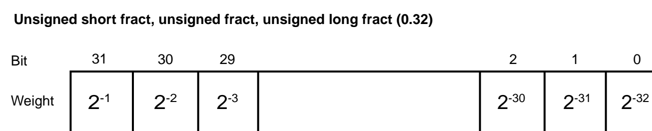

Figure 2-6: Data Storage Format for unsigned short fract, unsigned fract, and unsigned long fract (0.32)

Therefore, to represent 0.25 in `unsigned fract`, the hexadecimal representation is `0x40000000` (2^{-2}). `unsigned short fract`, `unsigned fract`, and `unsigned long fract` cannot represent +1 exactly, but they get quite close with `0xffffffff`.

Precision Restrictions With 40-Bit Floating-Point Arithmetic

CCES does not provide full support for 40-bit arithmetic for SHARC processors. If you attempt to use 40-bit arithmetic, you may encounter circumstances that result in variable precision. For instance:

- By default, the SHARC memory is configured for 32-bit data, so any values that are saved and restored from memory will lose precision.
- The compiler often copies data in a way that does not preserve all 40 bits of data. For example, it may use a DAG register as temporary storage if this offers a performance benefit.

If you enable 40-bit memory accesses, it is recommended that you compile any C code with the `-extra-precision` compiler switch.

The following run-time library routines supplied with CCES use 40-bit arithmetic internally (their inputs and outputs do not):

asinf	cabsf	cartesianf	cexpf

cfft_mag	cosf	div	fir (the scalar-valued version from the header file filters.h)
fmodf	gen_blackman	gen_hamming	gen_hanning
gen_harris	iir (the scalar-valued version from the header file filters.h)	ldiv	normf
polarf	rfft_mag	rmsf	rqsrtf
sinf	sqrtf	twidfftf	twidfft

If the switch -double-size-64 has not been specified, then the following functions also use 40-bit arithmetic internally:

asin	cabs	cexp	cos
fmod	norm	polar	rms
rsqrt	sin	sqrt	

The compiler support routines for the following C/C++ operations use 40-bit arithmetic:

- Long double-to-unsigned integer conversion
- Modulus operator
- Integer division

To preserve all 40 bits of registers used in these routines, we recommend using the interrupt dispatchers provided by Analog Devices; refer to the *System Run-Time Documentation* in the CCES online help.

Memory Section Usage

The C/C++ run-time environment requires that a specific set of memory section names are used to place code in memory. In assembly language files, these names are used as labels for the .SECTION directive. In the .ldf file, these names are used as labels for the output section names within the SECTIONS{} command. For information on .ldf file syntax and other information on the linker, see the *Linker and Utilities Manual*.

The ***Input Section Names Used by the Compiler and Libraries*** table lists the input section names used by the compiler and run-time libraries. Refer to this table when creating your own mappings in an .ldf file. You can change the section into which the compiler maps your code or data by using #pragma section/#pragma default_section.

Table 2-58: Input Section Names Used by the Compiler and Libraries

Names	*Usage Description*
seg_pmco	This section must be in Program Memory (PM), holds code, and is required by some functions in the C/C++ run-time library. For more information, see Code Storage in Program Memory.

Table 2-58: Input Section Names Used by the Compiler and Libraries (Continued)

Names	*Usage Description*
`seg_swco`	This section contains short-word instructions for targets that support VISA (variable instruction set) execution. It is used by the compiler and by some functions in the short-word variants of the C/C++ run-time libraries. For more information, see Code Storage in Program Memory.
`seg_dmda`	This section must be in Data Memory (DM), 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 Storage in Data Memory. For processors that support byte-addressing, this section can be either 32-bit or 8-bit in width, though can only be used with a single width within each assembly file.
`seg_dmda_nw`	This section is only used on processors that support byte-addressing. It is identical to `seg_dmda`, but is only for use with 32-bit width.
`seg_dmda_bw`	This section is only used on processors that support byte-addressing. It is identical to `seg_dmda`, but is only for use with 8-bit width.
`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 Data Storage in Program Memory. For processors that support byte-addressing, this section can be either 32-bit or 8-bit in width, though can only be used with a single width within each assembly file.
`seg_pmda_nw`	This section is only used on processors that support byte-addressing. It is identical to `seg_pmda`, but is only for use with 32-bit width.
`seg_pmda_bw`	This section is only used on processors that support byte-addressing. It is identical to `seg_pmda`, but is only for use with 8-bit width.
`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.
`iv_code`	This section contains the interrupt vector code, and must be mapped to the correct interrupt address range. For more details on interrupt management, refer to the *System Run-Time Documentation*.
`seg_int_code` `seg_int_code_sw`	These input sections are used to ensure that code is mapped to internal memory, and never to external memory, when it is desirable for performance reasons or necessary for any other reason (such as a silicon anomaly workaround) . The `seg_int_code` section can be used to map normal-word or short-word code on processors that support VISA execution. The `seg_int_code_sw` input section can only be used to map short-word code. The default and generated `.ldf` files map these sections to internal memory configured for code of the required width. The run-time libraries make use of `seg_int_code` to avoid silicon anomalies 02000055 and 04000046 in a small number of functions.

The following sections cover:

- Code Storage in Program Memory
- Data Storage in Data Memory
- Data Storage in Program Memory
- Run-Time Stack Storage
- Run-Time Heap Storage

- Initialization Data Storage

Code Storage in Program Memory

For processors that do not support VISA execution, `seg_pmco` is the location where the compiler puts all the instructions that it generates when you compile your program. When linking, use your `.ldf` file to map this section to a program memory (PM) section.

On processors that support VISA execution, the compiler puts all the instructions that it generates into `seg_swco` by default. When linking, use your `.ldf` file to map this section to an SW-qualified memory output section. When `-nwc` or `-normal-word-code` is used, the compiler puts all instructions into `seg_pmco`. When linking, use your `.ldf` file to map this section to a PM-qualified output section.

If you are assembling legacy assembly files and are using VISA execution support in your executable, use your `.ldf` file to map the input sections to an SW-qualified output section.

For processors that support VISA execution, the run-time libraries use both `seg_pmco` and `seg_swco` section names for VISA code; when mapping input sections to a VISA output section, the linker only maps input sections with the "short word" qualifier.

Data Storage in Data Memory

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 places static and global 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). The following example allocates an array of 10 integers in the DM data section:

```
static int data [10];
```

Data Storage in Program Memory

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 and global 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). The following example allocates an array of 10 integers in the PM data section:

```
static int pm coeffs[10];
```

Run-Time Stack Storage

Because the run-time environment cannot function without a stack, you must define one in DM space. A typical size for the run-time stack is 8K 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.

Stack space is not provided by an input section. Instead, the `.ldf` file uses the `RESERVE` and `RESERVE_EXPAND` directives to reserve a region of space within data memory. Depending on the target processor and the link-time options, the stack may be allocated a region of memory by itself, or a single, contiguous region may be allocated for both the stack and the default heap, with the stack occupying the higher addresses and the heap occupying the lower addresses.

Run-Time Heap Storage

To dynamically allocate and deallocate memory at runtime, the C/C++ run-time library includes several functions: `malloc`, `calloc`, `realloc` and `free`. These functions allocate memory from the run-time heap by default. A typical size for the run-time heap is 16K 32-bit words of data memory.

The default heap is not provided by an input section. Instead, the `.ldf` file uses the `RESERVE` and `RESERVE_EXPAND` directives to reserve a region of space within data memory. Depending on the target processor and the link-time options, the default heap may be allocated a region of memory by itself, or a single, contiguous region may be allocated for both the stack and the default heap, with the stack occupying the higher addresses and the heap occupying the lower addresses.

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 for more information on the use of multiple heaps.

NOTE: A default heap is always required by every project, as the run-time library makes some use of it internally.

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`.

- When using the `elfloader` utility, the `seg_init` section needs only 16 slots/locations of space.
- When using the `mem21k` utility, all RAM memory initialization will be stored in the `seg_init` PM ROM section, so the seg_init section will need to be sized accordingly.

For more information, see Memory Initialization.

Global Array Alignment

Global arrays must be aligned on a 64-bit word boundary or greater; the compiler will normally use this knowledge when optimizing accesses. If you declare arrays in assembly files that will be accessed from C/C++, use the `.ALIGN` directive to ensure the array's starting address has an alignment of 2 words or greater.

Controlling System Heap Size and Placement

The system heap is the default heap used by calls to allocation functions like `malloc()` in C and the new operator in C++. System heap placement and size are specified in the application's `.ldf` file.

For details on adding and managing additional heaps besides the system heap, see Using Multiple Heaps.

The following sections cover:

- Managing the System Heap in the IDE
- Managing the System Heap in the .LDF File
- Standard Heap Interface

Managing the System Heap in the IDE

The `.ldf` files created by the ***Project Wizard***, with the ***Startup Code/LDF*** option accepted, can be controlled by the ***System Configuration Overview*** dialog box.

1. Expand your new project in a project navigation view, such as ***Project Explorer***.
2. Double-click `system.svc`.

   The ***Startup Code/LDF*** component appears in the ***System Configuration Overview*** dialog box.
3. Click the ***Startup Code/LDF*** tab at the bottom of the dialog box.
4. Click the ***LDF*** tab that appears at the left of the dialog box.

   The ***LDF Configuration*** page appears.
5. In the ***System heap*** area, enable ***Customize the system heap***.
6. (Optional) Modify the size and choose memory for the system heap.
7. When you have modified the settings as required, save the changes, via ***Ctrl+S***, ***File*** > ***Save***, or by clicking the floppy disk icon in the toolbar. This causes the IDE to generate an updated LDF and related startup-code files, which configure your heaps during the application's startup.

Managing the System Heap in the .LDF File

If an `.ldf` file has not been added to the project either by using the Project Wizard or by using a custom file, a default `.ldf` file from the `<install_path>\SHARC\ldf` directory will be used.

By default, the compiler uses the file `arch.ldf`, where arch is specified via the `-proc` *`arch`* switch. For example, if `-proc ADSP-21469` is used, the compiler defaults to using `adsp-21469.ldf`. The entry controlling the heap for processors without support for byte-addressing has a format similar to the following (simplified for clarity):

```
// macro that defines minimum system heap size
#define HEAP_SIZE 16K
dm_block3_dm_data_prio0
{
```

```
   INPUT_SECTION_ALIGN(2)
   // allocate minimum of HEAP_SIZE to system heap
   RESERVE(sys_heap, sys_heap_length = HEAP_SIZE, 2)
}> mem_block3_dm32

// all other uses of mem_block3_dm32
sys_heap
{
   INPUT_SECTION_ALIGN(2)
   // if any of mem_block3_dm32 is unused, add to system heap
   RESERVE_EXPAND(sys_heap, sys_heap_length, 0)
   // define symbols to configure the heap for runtime support
   ldf_heap_space = sys_heap;
   ldf_heap_end = ldf_heap_space + sys_heap_length;
   ldf_heap_length = ldf_heap_end - ldf_heap_space;
}> mem_block3_dm32
```

In this example, the minimal size of the heap can be modified by changing the definition of the `HEAP_SIZE` macro. If this value is larger than the memory output section being used, the linker issues error `li2040`.

Processors with support for byte-addressing (see the the ***Processors Supporting Byte-Addressing*** table in Processor Features) have a slightly different format for the heap entry. The heap size is still specified in 32-bit words, but the output section is filtered by the DM qualifier:

```
// macro that defines minimum system heap size
#define HEAP_SIZE 16K
dm_block3_dm_data_prio0 DM
{
   INPUT_SECTION_ALIGN(2)
   // allocate minimum of HEAP_SIZE to system heap
   RESERVE(sys_heap, sys_heap_length = HEAP_SIZE, 2)
}> mem_block3_bw

// all other uses of mem_block3_bw
sys_heap DM
{
   INPUT_SECTION_ALIGN(2)
   // if any of mem_block3_bw is unused, add to system heap
   RESERVE_EXPAND(sys_heap, sys_heap_length, 0)
   // define symbols to configure the heap for runtime support
   ldf_heap_space = sys_heap;
   ldf_heap_end = ldf_heap_space + sys_heap_length;
   ldf_heap_length = ldf_heap_end - ldf_heap_space;
}> mem_block3_bw
```

The following macros can be used to configure the sizes of the system heap and stack, when using the default `.ldf` files. When using these macros, all three must be defined, for any of the definitions to take effect. All values are specified in 32-bit words.

- `HEAP_SIZE` – Defines the size of the system heap. A typical value would be “7K”.

- `STACK_SIZE` – Defines the size of the system stack. A typical value would be “8K”.
- `STACKHEAP_SIZE` – Defines the size of the combined area used for system heap and system stack. A typical value would be “15K”. Must be defined to be the sum of `HEAP_SIZE` and `STACK_SIZE`.

The default `.ldf` files support the placement of heaps in `L1` or SDRAM (where available). By default, `L1` is used. To select alternative heap placement, the following macro can be defined when linking:

- `USE_SDRAM_HEAP` – Causes SDRAM memory to be used for the system heap. It provides large capacity but is slow to access. Enabling data cache for the memory used reduces the performance impact.

Besides the default system heap, you can also define other heaps. See Using Multiple Heaps for more information.

Standard Heap Interface

The standard functions, `calloc` and `malloc`, allocate a new object from the default heap. If `realloc` is called with a null pointer, it too allocates a new object from the default heap. Previously allocated objects can be deallocated with the `free` or `realloc` functions. When a previously allocated object is resized with `realloc`, the returned object is in the same heap as the original object.

The `space_unused` function returns the number of bytes unallocated in the heap with index 0. Note that you may not be able to allocate all of this space due to heap fragmentation and the overhead that each allocated block needs.

Using Multiple Heaps

The C/C++ run-time library supports the standard heap management functions `calloc`, `free`, `malloc`, and `realloc`. By default, a single heap, called the *default heap*, serves all allocation requests that do not explicitly specify an alternative heap. The default heap is defined in the standard linker description file and the run-time header.

Any number of additional heaps can be defined. These heaps serve allocation requests that are explicitly directed to them. These additional heaps can be accessed via the extension routines `heap_calloc`, `heap_free`, `heap_malloc`, and `heap_realloc`. For more information, see Using the Alternate Heap Interface.

Multiple heaps allow the programmer to serve allocations using fast-but-scarce memory or slower-but-plentiful memory as appropriate.

The following sections cover:

- Defining a Heap
- Defining Additional Heaps in the IDE
- Defining Heaps at Runtime
- Tips for Working With Heaps
- Allocating C++ STL Objects to a Non-Default Heap
- Using the Alternate Heap Interface
- Freeing Space

Defining a Heap

Heaps can be defined in the IDE or at runtime. In both cases, a heap has three attributes:

- Start (base) address (the lowest usable address in the heap)
- Length (in words)
- User identifier (`userid`, a number >= 1)

The default system heap, defined at link-time, always has `userid 0`. In addition, heaps have indices. This is like the `userid`, except that the index is assigned by the system. All the allocation and deallocation routines use heap indices, not heap user IDs. A `userid` can be converted to its index using `heap_lookup()`. Be sure to pass the correct identifier to each function.

Defining Additional Heaps in the IDE

The Startup Code/LDF Add-in allows you to configure and extend your heaps through a convenient graphical interface:

- Modify the size of your heaps.
- Change whether they are in internal or external memory (where available).
- Add additional heaps, or remove them.

To add a new heap:

1. Expand your new project in a project navigation view, such as ***Project Explorer***.
2. Double-click `system.svc`.

   The ***Startup Code/LDF*** component appears in the ***System Configuration Overview*** dialog box.
3. Click the ***Startup Code/LDF*** tab at the bottom of the dialog box.
4. Click the ***LDF*** tab that appears at the left of the dialog box.

   The ***LDF Configuration*** page appears.
5. In the ***Stack and Heaps*** area, click ***System*** heap.
6. Click ***Add***.

   The ***Add User Heap*** dialog box appears. Fill in the details for your new heap.
7. Click ***OK***.
8. When you have modified the settings as required, save the changes, via ***Ctrl+S***, ***File*** > ***Save***, or by clicking the floppy disk icon in the toolbar. This causes the IDE to generate an updated LDF and related startup-code files, which configure your heaps during the application's startup.

The same interface allows you to edit additional heaps or remove them, via the ***Edit*** and ***Remove*** buttons, respectively.

Defining Heaps at Runtime

Heaps may also be defined and installed at runtime, using the `heap_install()` function:

```
int heap_install(void *base, size_t length, int userid);
```

This function can take any section of memory and start using it as a heap. It returns the heap index allocated for the newly installed heap, or a negative value if there was some problem. See Tips for Working With Heaps.

Reasons why `heap_install()` may return an error status include, but are not limited to:

- A heap using the specified `userid` already exists
- A new heap appears too small to be usable (length too small)

A heap is automatically initialized during installation. If necessary, a heap can be re-initialized later on. For more information, see Freeing Space.

Tips for Working With Heaps

Not all memory in a heap is available to users. A few words are reserved per heap and per allocation (rounded to ensure the allocation is two-word aligned) which are used for housekeeping. Thus, a heap of 256 words is unable to serve four blocks of 64 words.

Memory reserved for housekeeping precedes the allocated blocks. Thus, if a heap begins at `0x0080 0000`, this particular address is never returned to the user program as the result of an allocation request; the first request returns an address some way into the heap.

The base address of a heap must be appropriately aligned for a two-word memory access. This means that allocations can then be used for vector and long-word operations.

For C++ compliance, calls to `malloc` and `calloc` with a size of 0 will allocate a block of size 1.

Allocating C++ STL Objects to a Non-Default Heap

C++ STL objects can be placed in a non-default heap through use of a custom allocator. To do this, you must first create your custom allocator. Below is an example custom allocator that you can use as a basis for your own. The most important part of `customalloc.h` in most cases is the allocate function, where memory is allocated to the STL object.

Currently, the pertinent line of code assigns to the default heap (0):

```
Ty* ty = (Ty*) heap_malloc(0, n * sizeof(Ty));
```

Simply by changing the first parameter of `heap_malloc()`, you can allocate to a different heap:

- 0 is the default heap
- 1 is the first user heap
- 2 is the second user heap
- And so on

Once you have created your custom allocator, you must inform your STL object to use it. Note that the standard definition for "list":

```
list<int> a;
```

is the same as writing:

```
list<int, allocator<int> > a;
```

where "allocator" is the default allocator. Therefore, we can tell list "a" to use our custom allocator as follows:

```
list<int, customallocator<int> > a;
```

Once created, the list "a" can be used as normal. Also, `example.cpp` (below) is a simple example that shows the custom allocator being used.

customalloc.h

```
template <class Ty>
class customallocator {
public:
   typedef Ty value_type;
   typedef Ty* pointer;
   typedef Ty& reference;
   typedef const Ty* const_pointer;
   typedef const Ty& const_reference;

   typedef size_t size_type;
   typedef ptrdiff_t difference_type;

   template <class Other>
   struct rebind { typedef customallocator<Other> other; };
   pointer address(reference val) const { return &val; }
   const_pointer address(const_reference val)
                 const { return &val; }
   customallocator(){}
   customallocator(const customallocator<Ty>&){}
   template <class Other>
   customallocator(const customallocator<Other>&) {}
   template <class Other>
   customallocator<Ty>& operator=(const customallocator&)
                 { return (*this); }

   pointer allocate(size_type n, const void * = 0) {
      Ty* ty = (Ty*) heap_malloc(0, n * sizeof(Ty));
      cout << "Allocating 0x" << ty << endl;
      return ty;
   }

   void deallocate(void* p, size_type) {
      cout << "Deallocating 0x" << p << endl;
      if (p) free(p);
   }
```

```
   void construct(pointer p, const Ty& val)
                { new((void*)p)Ty(val); }
   void destroy(pointer p) { p->~Ty(); }
   size_type max_size() const { return size_t(-1); } };
```

example.cpp

```
#include <iostream>
#include <list>
using namespace std;
#include <customalloc.h>    // include your custom allocator

main(){
   cout << "creating list" << endl;
   list <int, customallocator<int> > a;
                  // create list with custom allocator
   cout.setf(ios_base::hex,ios_base::basefield);
   cout << "pushing some items on the back" << endl;
   a.push_back(0xaaaaaaaa);   // push items as usual
   a.push_back(0xbbbbbbbb);
   while(!a.empty()){
      cout << "popping:0x" << a.front() << endl;
                           //read item as usual
      a.pop_front();       //pop items as usual
   }
   cout << "finished." << endl;
}
```

Using the Alternate Heap Interface

The C run-time library provides the alternate heap interface functions `heap_calloc`, `heap_free`, `heap_malloc`, and `heap_realloc`. These routines work in exactly the same way as the corresponding standard functions without the `heap_` prefix, except that they take an additional argument that specifies the heap index.

These are the library functions that can be used to initialize the heaps, to allocate memory, and to free memory; the functions are described in the *C/C++ Library Manual for SHARC Processors*.

```
int heap_install(void base, size_t length, int userid);
int heap_init(int idx);
void *heap_calloc(int idx, size_t nelem, size_t elsize)
void *heap_free(int idx, void *)
void *heap_malloc(int idx, size_t length)
void *heap_realloc(int idx, void *, size_t length)
int heap_space_unused(int idx);
```

The actual entry point names for the alternate heap interface routines have an initial underscore. The `stdlib.h` standard header file defines equivalent prototypes without the leading underscores, which are mapped onto the library entry points using #pragma linkage_name *identifier*.

Note that for

```
heap_realloc(idx, NULL, length)
```

the operation is equivalent to

```
heap_malloc(idx, length)
```

However, for

```
heap_realloc(idx, ptr, length)
```

where `ptr != NULL`, the supplied `idx` parameter is ignored; the reallocation is always done from the heap from which `ptr` was allocated.

Similarly,

```
heap_free(idx, ptr)
```

ignores the supplied index parameter, which is specified only for consistency-the space indicated by `ptr` is always returned to the heap from which it was allocated.

The `heap_space_unused(int idx)` function returns the number of words unallocated in the heap with index `idx`. The function returns `-1` if there is no heap with the requested heap index.

C++ Run-Time Support for the Alternate Heap Interface

The C++ run-time library provides support for allocation and release of memory from an alternative heap via the `new` and `delete` operators.

Heaps should be initialized with the C run-time functions as described. These heaps can then be used via the `new` and `delete` mechanism by simply passing the heap ID to the `new` operator. There is no need to pass the heap ID to the `delete` operator as the information is not required when the memory is released.

The routines are used as in the example below.

```
#include <heapnew>

char *alloc_string(int size, int heapID)
{
     char *retVal = new(heapID) char[size];
     return retVal;
}

void free_string(char *aString)
{
     delete aString;
}
```

Freeing Space

When space is "freed", it is not returned to the "system". Instead, freed blocks are maintained on a free list within the heap in question. The blocks are coalesced where possible.

It is possible to re-initialize a heap, emptying the free list and returning all the space to the heap itself, using the `heap_init` function:

```
int heap_init(int index)
```

This returns zero for success, and nonzero for failure. Note, however, that this discards all records within the heap, so it may not be used if there are any live allocations on the heap still outstanding.

Startup and Termination

When the processor starts running, it has to transfer control to the application's `main()` function, but before doing so it has to ensure that all the expected parts of the C/C++ run-time environment have been set up, including:

- Registers, which must be configured according to the rules in Registers.
- Heap and stack, which must be set up according to Controlling System Heap Size and Placement and Managing the Stack.
- Global variables must have been initialized to their starting values.
- Constructors of any static global instances must have been run.
- The arguments to `main()`, `argc`, and `argv`, must have been set up.

This is the job of the startup code (or "C Run-Time Header", or "CRT"). The startup code is described in the *System Run-Time Documentation*, but some additional information is provided in the following sections:

- Memory Initialization
- Global Constructors
- Support for argv/argc

Memory Initialization

When control flow reaches the start of `main()`, global and static variables must have been initialized to their default values. When you build your application, the toolchain arranges for the executable image to contain sections of memory that are either zero- or value-filled, depending on how your data is declared. The image also contains sections that are filled with executable code. Further details are in Memory Section Usage.

During development, when you load your application into your processor using the IDE, the IDE copies the contents of those sections from your executable image into the processor's memory.

Once your application is complete, you have to change your application so that you no longer rely on using the IDE to load it into memory. This can be done by either:

- Creating a bootable image with the loader; see Bootable Images.
- Creating a non-bootable image with the memory initializer; see Non-Bootable Images.

Bootable Images

Usually, you will use the `elfloader` utility to create a bootable image that can be stored in non-volatile memory, such as a SPI flash, and loaded into memory at power-up by the Boot Code. In this model, the Boot Code arranges

for all of your application's code and data sections to be copied into the final volatile memory space before control is transferred to your application.

The elfloader utility processes your executable file, producing a 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.

For details on this process, refer to the *Loader and Utilities Manual*, and to your processor's programming reference manual.

Non-Bootable Images

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.

In this model, when the Boot Code transfers control to your application, your application's code and data have not yet all been transferred to their final locations in volatile memory. Instead, the startup code (which is in non-volatile memory) invokes the __lib_setup_memory run-time library function, which processes the initialization stream. This performs the task of transferring your application's code and data to volatile memory.

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_memory routine that is called from the run-time header.

For more details, refer to the *System Run-Time Documentation* and *Linker and Utilities Manual*.

Global Constructors

This section covers:

- Constructors and Destructors of Global Class Instances
- Constructors, Destructors and Memory Placement

Constructors and Destructors of Global Class Instances

Constructors for global class instances are invoked by the C/C++ run-time header during start-up. There are several components that allow this to happen:

- The associated data space for the instance
- The associated constructor (and destructor, if one exists) for the class
- A compiler-generated "start" routine
- A compiler-generated table of such "start" routines
- A compiler-constructed linked-list of destructor routines
- The run-time header itself

The interaction of these components is as follows.

The compiler generates a "start" routine for each module that contains globally-scoped class instances that need constructing or destructing. There is at most one "start" routine per module; it handles all the globally-scoped class instances in the module:

- For each such instance, it invokes the instance's constructor. This may be a direct call, or it may be inlined by the compiler optimizer.
- If the instance requires destruction, the "start" routine registers this fact for later, by including pointers to the instance and its destructor into a linked list.

The start routine is named after the first such instance encountered, though the classes are not guaranteed to be constructed or destructed in any particular order (with the exception that destructors are called in the reverse order of the constructors). Such instances should not have any dependency on construction order; the `-check-init-order` switch is useful for verifying this during system development, as it plants additional code to detect uses of unconstructed objects during initialization.

A pointer to the "start" routine is placed into the `ctdm` section of the generated object file. When the application is linked, all `ctdm` sections are mapped into the same `ctdm` output section, forming a table of pointers to the "start" routines. An additional `ctdml` object is appended to the end of the table; this contains a terminating NULL pointer.

When the run-time header is invoked, it calls `_ctor_loop()`, which walks the table of `ctdm` sections, calling each pointed-to "start" function until it reaches the NULL pointer from `ctdml`. In this manner, the run-time header calls each global class instance's constructor, indirectly through the pointers to "start" functions. On processors that support byte-addressing (see the ***Processors Supporting Byte-Addressing*** table in Processor Features), constructors for global class instances in source files compiled with `-char-size-8` are called before those from source files compiled with `-char-size-32`.

When the program reaches `exit()`, either by calling it directly or by returning from `main()`, the `exit()` routine follows the normal process of invoking the list of functions registered through the `atexit()` interface. One of these is a function that walks the list of destructors, invoking each in turn (in reverse order from the constructors).

This function is registered with `atexit()` during the run-time header, before `main()` is called.

NOTE: Functions registered with `atexit()` may not make reference to global class instances, as the destructor for the instance may be invoked before the reference is used.

Constructors, Destructors and Memory Placement

By default, the compiler places the code for constructors and destructors into the same section as any other function's code. This can be changed either by specifying the section specifically for the constructor or destructor (see #pragma section/#pragma default_section and Placement Support Keyword (section)), or by altering the default destination section for generated code (see #pragma section/#pragma default_section and `-section`*`id=section_name[, id=section_name...]`*). Note that if a constructor is inlined into the "start" routine by the optimizer, such placement will have no effect. For more information, see Inlining and Sections.

While normal compiler-generated code is placed into the `CODE` area, the "start" routine is placed into the `STI` area. Both `CODE` and `STI` default to the same section, but may be changed separately using `#pragma default_section` or the `-section` switch (as the "start" function is an internal function generated by the compiler, its placement cannot be affected by `#pragma section`).

The pointer to the "start" routine is placed into the `ctdm` section. This is not configurable, as the invocation process relies on all of the "start" routine pointers being in the same section during linking, so that they form a table. It is essential that all relevant `ctdm` sections are mapped during linking; if a `ctdm` section is omitted, the associated constructor will not be invoked during start-up, and run-time behavior will be incorrect.

If destructors are required, the compiler generates data structures pointing to the class instance and destructor. These structures are placed into the default variable-data section (the `DATA` area).

Support for argv/argc

By default, the facility to specify arguments that are passed to your `main()` (`argv/argc`) at run-time is enabled. However, to correctly set up `argc` and `argv` requires you to do some additional configuration. Modify your application as follows:

- Define your command-line arguments in C by defining a variable called `__argv_string`. When linked, your new definition overrides the default zero definition otherwise found in the C run-time library. For example,

  ```
  extern const char __argv_string[] = "-in x.gif -out y.jpeg";
  ```

  For processors which support byte-addressing, this definition should be compiled with the same `-char-size-8/-char-size-32` switch as the source file which contains `main`.

Compiler C++ Template Support

The compiler provides C++ templates as defined in the ISO/IEC 14882:2003 C++ standard.

Template Instantiation

Templates are instantiated automatically by the prelinker during compilation (see Compiler Components). 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.

Exported Templates

The compiler supports the `export` keyword. An exported template does not need to be present in a translation unit that uses the template. For example, the following is a valid C++ program consisting of two translation units:

```
// File 1
#include <iostream>
static void print(void) { std::cout << "File 1" << std::endl;}
export template <class T> T const &maxii(T const &a, T const &b);
int main()
{
  print();
  return maxii(7,8);
}

// File 2
#include <iostream>
static void print(void) { std::cout << "File 2" << std::endl;}
export template <class T> T const &maxii(T const &a, T const &b)
{
  print();
  return (a>b) ? a : b;
}
```

The first file makes use of the `maxii()` function exported by the second. Unrelated to this, both files declare their own, private copy of the `print()` function.

The two files are separate translation units; one is not included in the other, so no linking errors arise due to the individual definitions of the `print()` functions. If `file1.c` obtained `file2.c`'s definition of `maxii()` by including `file2.c` into file1.c (whether explicitly or implicitly - see Implicit Instantiation), file1.c would also include `file2.c`'s definition of the `print()` function, leading to a linkage error.

When a file containing a definition of an exported template is compiled, a file with a `.et` suffix is created and some extra information is included in the associated `.ti` file. The `.et` files are used by the compiler to find the translation units that define a given exported template.

Implicit Instantiation

As an alternative to Exported Templates, the compiler can use a method called *implicit instantiation*, which is common practice. It results in having both the specification and definition available at point of instantiation.

NOTE: Implicit instantiation does not conform to the ISO/IEC 14882:2003 C++ standard, and does not work with exported templates. Implicit instantiation is disabled by default. It can be enabled via the `-implicit-inclusion` switch.

Implicit instantiation involves placing template specifications in a header (for example, `.h`) file and the definitions in a source (for example, `.cpp`) file. Any file being compiled that includes a header file containing template specifications will instruct the compiler to implicitly include the corresponding `.cpp` file containing the definitions of the 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.

NOTE: Because the whole of the file is included, other definitions in the .cpp file will also be visible, which can lead to problems if the .cpp file contains definitions unrelated to the templates being instantiated. Exported Templates avoid this problem.

When generating dependencies, the compiler will only parse each implicitly included `.cpp` file once. This parsing avoids excessive compilation times in situations where a header file that implicitly includes a source file is included several times. If the `.cpp` file should be included implicitly more than once, the `-full-dependency-inclusion` switch can be used. For example, the file may contain macro guarded sections of code. This may result in more time required to generate dependencies.

Generated Template Files

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 an `.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 Un-Instantiated Templates

If the prelinker is unable to instantiate all the templates required for a particular link, 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]'
referenced
                from '.\Debug\main.doj'
                'T1 *Buffer<T1>::_getAddress() const [with T1=Complex<short>]
                [_getAddress__33Buffer__tm__19_16Complex__tm__2_sCFv_PZ1Z]'
referenced
                from '.\Debug\main.doj'
                'T1 Complex<T1>::_getReal() const [with T1=short]
                [_getReal__16Complex__tm__2_sCFv_Z1Z]' referenced from
                '.\Debug\main.doj'
Linker finished with 1 error
```

Careful examination of the linker errors reveals which instantiations have not been made. Below are some examples.

```
Missing instantiation:
    Complex<short> Complex<short>::conjugate()
Linker Text:
   'Complex<T1> Complex<T1>::_conjugate() const [with T1=short]
   [_conjugate__16Complex__tm__2_sCFv_18Complex__tm__4_Z1Z]'
   referenced from '.\Debug\main.doj'

Missing instantiation:
   Complex<short> *Buffer<Complex<short>>::getAddress()
Linker Text:
   'T1 *Buffer<T1>::_getAddress() const [with T1=Complex<short>]
   [_getAddress__33Buffer__tm__19_16Complex__tm__2_sCFv_PZ1Z]'
   referenced from '.\Debug\main.doj'

Missing instantiation:
   Short Complex<short>::getReal()
Linker Text:
   'T1 Complex<T1>::_getReal() const [with T1=short]
   [_getReal__16Complex__tm__2_sCFv_Z1Z]' referenced from
   '.\Debug\main.doj'
```

There could be many reasons for the prelinker being unable to instantiate these templates, but the most common is that the `.ti` and `.ii` files associated with an object file have been removed. Only source files that can contain instantiated templates will have associated `.ti` and `.ii` files, and without this information, the prelinker may not be able to complete its task. Removing the object file and recompiling will normally fix this problem.

Another possible reason for un-instantiated templates at link time is when implicit inclusion (described above) is disabled but the source code has been written to require it. Explicitly compiling the `.cpp` files that would normally have been implicitly included and adding them to the final link is normally all that is needed to fix this.

Another likely reason for seeing the linker errors above is invoking the linker directly. It is the compiler's responsibility to instantiate C++ templates, and this is done automatically if the final link is performed via the compiler driver. The linker itself contains no support for instantiating templates.

File Attributes

A file attribute is a name-value pair that is associated with a binary object, whether in an object file (`.doj`) or in a library file (`.dlb`). One attribute name can have multiple values associated with it. Attribute names and values are strings. A valid attribute name consists of one or more characters matching the following pattern:

```
[a-zA-Z_][a-zA-Z_0-9]*
```

An attribute value is a non-empty character sequence containing any characters apart from NUL.

Attributes help with the placement of run-time library functions. All of the runtime library objects contain attributes which allow you to place time-critical library objects into internal (fast) memory. Using attribute filters in the LDF, you can place run-time library objects into internal or external (slow) memory, either individually or in groups.

This section describes:

- Automatically-Applied Attributes
- Default LDF Placement
- Sections Versus Attributes
- Using Attributes

Automatically-Applied Attributes

By default, the compiler automatically applies a number of attributes when compiling a C/C++ file.

For example, it applies the `Content`, `FuncName`, and `Encoding` attributes. These automatically-applied attributes can be disabled using the `-no-auto-attrs` switch. The ***Content Attributes*** figure shows a content attribute tree.

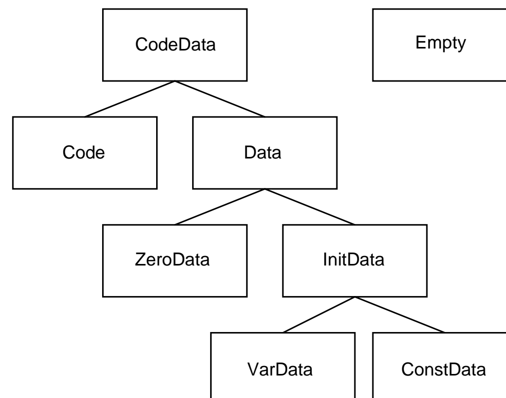

Figure 2-7: Content Attributes

Content Attributes

The `Content` attributes can be used to map binary objects according to their kind of content, as show by the ***Values of the Content Attribute*** table.

Table 2-59: Values of the Content Attribute

Value	*Description*
`CodeData`	This is the most general value, indicating that the binary object contains a mix of content types.
`Code`	The binary object does not contain any global data, only executable code. This can be used to map binary objects into program memory, or into read-only memory.
`Data`	The binary object does not contain any executable code. The binary object may not be mapped into dedicated program memory. The kinds of data used in the binary object vary.
`ZeroData`	The binary object contains only zero-initialized data. Its contents must be mapped into a memory section with the `ZERO_INIT` qualifier, to ensure correct initialization.
`InitData`	The binary object contains only initialized global data. The contents may not be mapped into a memory section that has the `ZERO_INIT` qualifier.
`VarData`	The binary object contains initialized variable data. It must be mapped into read-write memory, and may not be mapped into a memory section with the `ZERO_INIT` qualifier.
`ConstData`	The binary object contains only constant data (data declared with the C const qualifier). The data can be mapped into read-only memory (see also the `-const-read-write` switch and its effects).
`Empty`	The binary object contains neither functions nor global data.

FuncName Attributes

The `FuncName` attributes are multi-valued attributes whose values are all the assembler linkage names of the defined names in `obj`.

Encoding Attributes

The `Encoding` attributes can be used to map binary objects according to the encoding of code they contain, as shown by the ***Values of the Encoding Attribute*** table.

Table 2-60: Values of the Encoding Attribute

Value	*Description*
`SW`	The binary object contains only short-word code (processors that support VISA execution only).
`NW`	The binary object contains only normal-word code.
`Mixed`	The binary object contains a mixture of short-word and normal-word code (processors that support VISA execution only).

Default LDF Placement

The default .ldf file is written so that the order of preference for putting an object in section seg_dmda, seg_pmco or seg_swco depends on the value of the prefersMem attribute. Precedence is given in the following order:

1. Highest priority is given to binary objects that have a prefersMem attribute with a value of internal.
2. Next priority is given to binary objects that have no prefersMem attribute, or a prefersMem attribute with a value that is neither internal nor external.
3. Lowest priority is given to binary objects with a prefersMem attribute with the value external.

Although the default .ldf files only reference the values internal and external, prefersMem may have other values. For example, an object using a value such as L2 will be given second priority, as the value is neither internal nor external. You may modify your .ldf file to assign appropriate priority to any value you choose, by mapping objects with higher-priority before objects with lower-priority values.

The prefersMemNum attribute is similar to the prefersMem attribute, but is given numerical values instead of textual values. This makes it easier to assign priority when there are many different levels, because you can use relational comparisons in the .ldf file instead of just equalities and inequalities. The ***Values for prefersMemNum Attribute*** table shows the numerical values used by the run-time library for each corresponding prefersMem attribute value.

Table 2-61: Values for prefersMemNum Attribute

prefersMem Attribute Value	*prefersMemNum Attribute Value*
internal	30
any	50
external	70

Sections Versus Attributes

File attributes and section qualifiers (see Automatically-Applied Attributes) can be thought of as being somewhat similar, since they can both affect how the application is linked. There are important differences, however. These differences will affect whether you choose to use sections or file attributes to control the placement of code and data.

Granularity

Individual components-global variables and functions-in a binary object can be assigned different sections, then those section assignments can be used to map each component of the binary object differently. In contrast, an attribute applies to the whole binary object. This means you do not have as fine control over individual components using attributes as when using sections.

"Hard" Versus "Soft"

A section qualifier is a *hard* constraint: when the linker maps the object file into memory, it must obey all the section qualifiers in the object file, according to instructions in the `.LDF` file. If this cannot be done, or if the `.LDF` file does not give sufficient information to map a section from the object file, the linker will report an error.

With attributes, the mapping is *soft*: the default LDFs use the `prefersMem` attribute as a guide to give a better mapping in memory, but if this cannot be done, the linker will not report an error. For example, if there are more objects with `prefersMem=internal` than will fit into internal memory, the remaining objects will spill over into external memory. Likewise, if there are less objects with the attribute `prefersMem!=external` than are needed to fill internal memory, some objects with the `prefersMem=external` attribute may get mapped to internal memory.

Section qualifiers are rules that must be obeyed, while attributes are guidelines, defined by convention, that can be used if convenient and ignored if inconvenient. The `Content` attribute is an example: you can use the `Content` attribute to map `Code` and `ConstData` binary objects into read-only memory, if this is a convenient partitioning of your application. However, you need not do so if you choose to map your application differently.

Number of Values

Any given element of an object file is assigned exactly one section qualifier, to determine into which section it should be mapped. In contrast, an object file may have many attributes (or even none), and each attribute may have many different values. Since attributes are optional and act as guidelines, you need only pay attention to the attributes that are relevant to your application.

Using Attributes

You can add attributes to a file in two ways:

- Use #pragma file_attr("name[=value]"[, "name[=value]" [...]])
- Use the `-file-attr` *`name[=value]`* switch.

Refer to Example 1 and Example 2 on the use of attributes.

The run-time libraries have attributes associated with the objects in them. For more information on the attributes in run-time library objects, see ***Library Attributes*** in the *C/C++ Library Manual for SHARC Processors*.

Example 1

This example demonstrates how to use attributes to encourage the placement of library functions in internal memory.

Suppose the file `test.c` exists, as shown below:

```
#define MANY_ITERATIONS 500

void main(void) {
   int i;
   for (i = 0; i < MANY_ITERATIONS; i++) {
      fft_lib_function();
```

```
        frequently_called_lib_function();
    }
    rarely_called_lib_function();
}
```

Also suppose:

- The objects containing `frequently_called_lib_function` and `rarely_called_lib_function` are both in the standard library, and have the attribute `prefersMem=any`.
- There is only enough internal memory to map `fft_lib_function` (which has `prefersMem=internal`) and one other library function into internal memory.
- The linker chooses to map `rarely_called_lib_function` to internal memory.

For optimal performance in this example, `frequently_called_lib_function` should be mapped to the internal memory in preference to `rarely_called_lib_function`.

The `.ldf` file defines the following macro `$OBJS_LIBS_INTERNAL` to store all the objects that the linker should try to map to internal memory:

```
$OBJS_LIBS_INTERNAL =
      $OBJECTS{prefersMem("internal")},
      $LIBRARIES{prefersMem("internal")};
```

If they do not all fit in internal memory, the remainder get placed in external memory - no linker error will occur. To add the object that contains `frequently_called_lib_function` to this macro, extend the definition to read:

```
$OBJS_LIBS_INTERNAL =
      $OBJECTS{prefersMem("internal")},
      $LIBRARIES{prefersMem("internal")},
      $LIBRARIES{ libFunc("frequently_called_lib_function") };
```

This ensures that the binary object that defines `frequently_called_lib_function` is among those to which the linker gives highest priority when mapping binary objects to internal memory.

Note that it is not necessary for you to know which binary object defines `frequently_called_lib_function` (or even which library). The binary objects in the run-time libraries all define the `libFunc` attribute so that you can select the binary objects for particular functions without needing to know exactly where in the libraries a function is defined.

The modified line uses this attribute to select the binary object(s) for `frequently_called_lib_function` and append them to the `$OBJS_LIBS_INTERNAL` macro. The `.ldf` file maps objects in `$OBJS_LIBS_INTERNAL` to internal memory in preference to other objects. Therefore, `frequently_called_lib_function` gets mapped to L1.

Example 2

Suppose you want the contents of `test.c` to get mapped to external memory by preference. You can do this by adding the following pragma to the top of `test.c`:

```
#pragma file_attr("prefersMem=external")
```

or use the `-file-attr` switch on the following command line:

```
cc21k -file-attr prefersMem=external test.c
```

Both of these methods will mean that the resulting object file will have the attribute `prefersMem=external`. The `.ldf` files give objects with this attribute the lowest priority when mapping objects into internal memory, so the object is less likely to consume valuable internal memory space which could be more usefully allocated to another function.

NOTE: File attributes are used as guidelines rather than rules. If space is available in internal memory after higher-priority objects have been mapped, it is permissible for objects with `prefersMem=external` to be mapped into internal memory.

Implementation Defined Behavior

Each of the language standards supported by the compiler have implementation defined behavior for a list of areas. The implementation used by the compilers is detailed in this section.

Enumeration Type Implementation Details

The compiler by default implements the underlying type for enumerations as the first type from the following list that can be used to represent all the values in the specified enumeration: `unsigned int, int, unsigned long, long, unsigned long long, long long`. Enumeration constant values can be any integral type including `long long` and `unsigned long long`.

Enumerations types being implemented as `long long` or `unsigned long long` types is an Analog Devices extension to ANSI C89 standard (ISO/IEC 9899:1990). Allowing enumerations constants to be integral types other than `int` is an Analog Devices extension to the ANSI C89 and ANSI C99 (ISO/IEC 9899:1999) standards. These extensions can be disabled by using the `-enum-is-int` switch.

For more information, see `-enum-is-int`.

When `-enum-is-int` is used the compiler issues error cc0066 "`enumeration value is out of "int" range`" when it encounters enumeration constant values that cannot be held using an int type. Warning cc1661 "`enumeration value is greater than int type`" is issued when larger than int type enumeration values are used and not compiling with the `-enum-is-int` switch.

The different underlying types used by the compiler to implement enumerations can give rise to other compiler warnings. For example in the following enumeration the underlying type will be `unsigned int` which will result in warning cc0186 "`pointless comparison of unsigned integer with zero`".

```
typedef enum { v1, v2 } e1;
void check (e1 v) {
  if (v < 0) /* pointless comparison if e1 is unsigned */
      printf("out of range");
}
```

If a negative enumeration constant was added to the definition of e1 or if the example was compiled with the -`enum-is-int` switch the underlying type used will be signed int and there would be no warning issued for the comparison.

ISO/IEC 9899:1990 C Standard (C89 Mode)

The contents of this section refer to Annex G of the ISO/IEC 9899:1990 C Standard; subsection numbers such as 5.1.1.3 refer to the relevant section of that Standard, which has some implementation-defined aspect.

G3.1 Translation

5.1.1.3 How a diagnostic is identified

The compiler will emit descriptive diagnostics via the standard error stream at compilation time (e.g. "cc0223: function declared implicitly") or as annotations in generated assembly files.

G3.2 Environment

5.1.2.2.1 The semantics of the arguments to main

By default, `argv[0]` is a `NULL` pointer.

The values given to the strings pointed to be the `argv` argument can be defined by the user. For more information, see Support for argv/argc.

5.1.2.3 What constitutes an interactive device

An interactive device is considered a paired display screen and keyboard.

G3.3 Identifiers

6.1.2 The number of significant initial characters (beyond 31) in an identifier without external linkage

The number of significant initial characters in an identifier without external linkage is 15,000.

6.1.2 The number of significant initial characters (beyond 6) in an identifier with external linkage

Identifiers with external linkage are treated in the same way as identifiers without.

6.1.2 Whether case distinctions are significant in an identifier with external linkage

Case distinctions are significant.

G3.4 Characters

5.2.1 The members of the source and execution character sets, except as explicitly specified in this International Standard

The compiler supports the non-standard characters "$" and "`" (ASCII 39).

5.2.1.2 The shift states used for the encoding of multi-byte characters

No shift states are used for the encoding of multi-byte characters.

5.2.4.2.1 The number of bits in a character in the execution character set

When the `-char-size-8` switch is used on processors with support for byte-addressing (see the ***Processors Supporting Byte-Addressing*** table in Processor Features), a character is 8 bits in size. Otherwise it is 32 bits in size.

6.1.3.4 The mapping of members of the source character set (in character constants and string literals) to members of the execution character set

Characters in the source file are interpreted as ASCII values, which are also used in the execution environment.

6.1.3.4 The value of an integer character constant that contains a character or escape sequence not represented in the basic execution set or the extended character set for a wide character constant

An unrecognized escape sequence will have the escape character dropped. E.g. '\k' becomes 'k'.

6.1.3.4 The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multi-byte character

An integer character constant may contain 1 character. Using more than 1 character will result in warning cc2226 being issued and all but the last character being discarded.

Where a wide character contains more than one multi-byte character, only the first character is retained and warning cc0026 will be issued. Subsequent characters are discarded

6.1.3.4 The current locale used to convert multi-byte characters into corresponding wide characters (codes) for a wide character constant

Only the "C" locale is supported in Analog Devices toolchain and processors.

6.2.1.1 Whether a "plain" char has the same range of values as signed char or unsigned char

A "plain" `char` has the same range and value as a `signed char`.

G3.5 Integers

6.1.2.5 The representations and sets of values of the various types of integers

The representations are shown in the ***Representations of Integer Types for the ADSP-21xxx and ADSP-SCxxx Processors*** (with no support for byte-addressing, or when using the -char-size-32 switch) and ***Representations of Integer Types when Using the -char-size-8 Switch on ADSP-21xxx and ADSP-SCxxx Processors*** (with support for byte-addressing) tables.

Table 2-62: Representations of Integer Types on the ADSP-21xxx and ADSP-SCxxx Processors (with no support for byte-addressing, or when using the -char-size-32 switch)

Type	*Width*	*Minimum Value*	*Maximum Value*
(signed) `char`	32 bits	-2147483648	2147483647
`unsigned char`	32 bits	0	4294967295
(signed) `short`	32 bits	-2147483648	2147483647
`unsigned short`	32 bits	0	4294967295
(signed) `int`	32 bits	-2147483648	2147483647
`unsigned int`	32 bits	0	4294967295
(signed) `long`	32 bits	-2147483648	2147483647
`unsigned long`	32 bits	0	4294967295
(signed) `long long`	64 bits	-9223372036854775808	9223372036854775807
`unsigned long long`	64 bits	0	18446744073709551615

Table 2-63: Representations of Integer Types when Using the -char-size-8 Switch on the ADSP-21xxx and ADSP-SCxxx Processors (with support for byte-addressing)

Type	*Width*	*Minimum Value*	*Maximum Value*
(signed) `char`	8 bits	-128	127
`unsigned char`	8 bits	0	255
(signed) `short`	16 bits	-32768	32767
`unsigned short`	16 bits	0	65535
(signed) `int`	32 bits	-2147483648	2147483647
`unsigned int`	32 bits	0	4294967295
(signed) `long`	32 bits	-2147483648	2147483647
`unsigned long`	32 bits	0	4294967295
(signed) `long long`	64 bits	-9223372036854775808	9223372036854775807
`unsigned long long`	64 bits	0	18446744073709551615

6.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented

When converting an unsigned integer to a signed integer of equal length, the exact value of the unsigned integer will be copied to the signed integer. If the sign bit is set, this will result in a negative number.

When converting a signed integer to a smaller signed integer, the lower bits of the signed integer (of the size of the smaller signed integer) are copied to the smaller signed integer. If the top-most copied bit is set, this will result in a negative number.

6.3 The results of bitwise operations on signed integers

The results of the operations are shown in the ***Bitwise Operations on Signed Integers*** table.

Table 2-64: Bitwise Operations on Signed Integers

<table>
<tr><td>~</td><td>Same as unsigned integer</td></tr>
<tr><td><<</td><td>Same as unsigned integer</td></tr>
<tr><td>>></td><td>Will fill upper bits with ones if sign bit was originally set</td></tr>
<tr><td>&</td><td>Same as unsigned integer</td></tr>
<tr><td>^</td><td>Same as unsigned integer</td></tr>
<tr><td>|</td><td>Same as unsigned integer</td></tr>
</table>

6.3.5 The sign of the remainder on integer division

The sign of the remainder on integer division will be the same as the sign of the first operand of the remainder operation.

6.3.7 The result of a right shift of a negative-valued signed integral type

Right shifts will retain the sign bit on a signed integer. All other bitwise operations treat signed integers as unsigned.

G3.6 Floating-Point

6.1.2.5 The representations and sets of values of the various types of floating-point numbers

The representations and value ranges are:

- `float`
  - 32 bits (1 sign bit, 8 exponent bits, 32 mantissa bits)

    -3.4028234663852886E+38 to 3.4028234663852886E+38
- `double` (default setting)
  - 32 bits (1 sign bit, 8 exponent bits, 32 mantissa bits)

    -3.4028234663852886E+38 to 3.4028234663852886E+38
- `double` (when compiling with `-double-size-64`)
  - 64 bits (1 sign bit, 11 exponent bits, 52 mantissa bits)

    -1.797693134862315708e+308 to 1.797693134862315708e+308
- `long double`
  - 64 bits (1 sign bit, 11 exponent bits, 52 mantissa bits)

    -1.797693134862315708e+308 to 1.797693134862315708e+308

6.2.1.3 The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value

Round to nearest, ties to even.

6.2.1.4 The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number

Round to nearest, ties to even.

G3.7 Arrays and Pointers

6.3.3.4, 7.1.1 The type of integer required to hold the maximum size of an array-that is, the type of the sizeof operator, size_t

`long unsigned int`.

6.3.4 The result of casting a pointer to an integer or vice-versa

A cast from pointer to integer results in the most-significant bits being discarded if the size of the pointer is larger than the integer. If the pointer is smaller than the integer type being cast to, the integer will be zero extended.

A cast from integer to pointer results in the most-significant bits being discarded if the size of the integer is larger that the pointer. If the integer is smaller than the pointer type being cast to, the pointer will be sign-extended.

6.3.6, 7.1.1 The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t

`long int`.

G3.8 Registers

6.5.1 The extent to which objects can actually be placed in registers by use of the register storage-class specifier

The `register` storage class specifier is ignored.

G3.9 Structures, Unions, Enumerations and Bit-Fields

6.3.2.3 A member of a union object is accessed using a member of a different type

The data stored in the appropriate location is interpreted as the type of the member accessed.

6.5.2.1 The padding and alignment of members of structures. This should present no problem unless binary data written by one implementation are read by another.

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

6.5.2.1 Whether a "plain" int bit-field is treated as a signed int bit-field or as an unsigned int bit-field

A "plain" `int` bit-field is treated as a `signed int` bit-field (including bit-fields of size 1).

6.5.2.1 The order of allocation of bit-fields within a unit

Low to High Order.

6.5.2.1 Whether a bit-field can straddle a storage-unit boundary

A bit-field will be placed in an adjacent storage unit instead of overlapping.

6.5.2.2 The integer type chosen to represent the values of an enumeration type

By default, the compiler defines enumeration types with integral types larger than `int`, if `int` is insufficient to represent all the values in the enumeration. The compiler can be forced to use only `int` through the use of the `-enum-is-int` switch.

G3.10 Qualifiers

6.5.3 What constitutes an access to an object that has volatile-qualified type

Any reference to a `volatile`-qualified object is considered to constitute an access.

G3.11 Declarators

6.5.4 The maximum number of declarators that may modify an arithmetic, structure, or union type

No maximum limit is enforced.

G3.12 Statements

6.6.4.2 The maximum number of case values in a switch statement

There is no hard-coded maximum number of `case` values in a `switch` statement.

G3.13 Preprocessing Directives

6.8.1 Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set; whether such a character constant may have a negative value

The character set used is the same.

Negative values are allowed.

6.8.2 The method for locating includable source files

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 (`-I directory[{,|;} directory...]`) in the order they are listed on the command line
3. Any directories on the standard list: `<install_path>\...\include`

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

6.8.2 The support of quoted names for includable source files

Quoted file names are supported.

6.8.2 The mapping of source file character sequences

The source file character sequence is mapped to its corresponding ASCII character sequence.

6.8.6 The behavior on each recognized #pragma directive

For more information, see Pragmas.

6.8.8 The definitions for __DATE__ and __TIME__ when respectively, the data and time of translation are not available

The macros `__DATE__` and `__TIME__` will be defined as "[date unknown]" and "[time unknown]" respectively.

G3.14 Library Functions

7.1.6 The null pointer constant to which the macro NULL expands

`NULL` expands to `((void *)0)`.

7.2 The diagnostic printed by and the termination behavior of the assert function

`ASSERT` [{failed assertion expression}] fails at `"{file name}":{line number}`

7.3.1 The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint, and isupper functions

The following characters are tested:

- `isalnum` - 0-9, a-z or A-Z
- `isalpha` - a-z or A-Z
- `iscntrl` - 0x00-0x1F or 0x7F
- `islower` - a-z
- `isprint` - 0x20-0x7E
- `isupper` - A-Z

7.5.1 The values returned by the mathematics functions on domain errors

The values are:

- `acos`: 0
- `asin`: 0
- `atan2`: 0
- `log`: `0`
- `log10`: 0

- `pow`: 0
- `sqrt`: 0
- `fmod`: 0

7.5.1 Whether the mathematics functions set the integer expression errno to the value of the macro ERANGE on underflow range errors

The state of `errno` should not be relied upon unless stated explicitly in the documentation.

7.5.6.4 Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero

Zero is returned.

7.7.1.1 The set of signals for the signal function

The following signals are supported:

- `SIGTERM`
- `SIGABRT`
- `SIGFPE`
- `SIGILL`
- `SIGINT`
- `SIGSEGV`

7.7.1.1 The semantics for each signal recognized by the signal function

After the handler is invoked, the disposition of the signal is not reset to `SIG_DFL`.

7.7.1.1 The default handling and the handling at program startup for each signal recognized by the signal function

By default, `SIGABRT` will cause the program to terminate. All other signals are ignored by default.

7.7.1.1 If the equivalent of signal(sig, SIG_DFL); is not executed prior to the call of a signal handler, the blocking of the signal that is performed

Blocking of signals is not performed prior to the call of the signal handler.

7.7.1.1 Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function

If the `SIGILL` signal is received, the reset to `SIG_DFL` is not performed.

7.9.2 Whether the last line of a text stream requires a terminating new-line character

The last line should have a terminating new-line character.

7.9.2 Whether space characters that are written out to a text stream immediately before a new-line character appear when read in

The space characters will appear.

7.9.2 The number of null characters that may be appended to data written to a binary stream

Any number of null characters may be appended.

7.9.3 Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file

End of the file.

7.9.3 Whether a write on a text stream causes the associated file to be truncated beyond that point

The file will become truncated.

7.9.3 The characteristics of file buffering

`stderr` is unbuffered, `stdio` is line-buffered, and other streams are fully buffered.

7.9.3 Whether a zero-length file actually exists

A zero-length file does exist.

7.9.3 The rule for composing valid file names

Any basic ASCII character that isn't reserved by the file system is valid.

7.9.3 Whether the same file can be open multiple times

A file can be opened multiple times.

7.9.4.1 The effect of the remove function on an open file

There will be no effect on the file and the function will return `-1`.

7.9.4.2 The effect if a file with the new name exists prior to a call to the rename function

There will be no effect on the files and the function will return `-1`.

7.9.6.1 The output for %p conversion in the fprintf function

The pointer address will be printed as an 8-character hexadecimal value. e.g. 00004010.

7.9.6.2 The input for %p conversion in the fscanf function

All valid values that can be interpreted as a hexadecimal value will be read until an invalid value or line break is reached, at which point no further characters are read. if the value is larger than can be stored in an 8-character hexadecimal, then the value will saturate.

7.9.6.2 The interpretation of a - character that is neither the first nor the last character in the scanlist for %[conversion in the fscanf function

A hyphen does not infer an inclusive range of values. e.g. `%[0-9]` will look for a sequence of '0', '-' and '5' chars.

7.9.9.1, 7.9.9.4 The value to which the macro errno is set by the fgetpos or ftell function on failure

`errno` should never be relied upon.

7.9.10.4 The messages generated by the perror function

`errno` should never be relied upon, so the error messages returned by this function should not be relied upon.

7.10.3 The behavior of the calloc, malloc, or realloc function if the size requested is zero

This is equivalent to a size request of 1.

7.10.4.1 The behavior of the abort function with regard to open and temporary files

`abort` will cause execution to jump to `exit` as if the program had run to the end of `main`.

7.10.4.3 The status returned by the exit function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE

The `exit` function never returns.

7.10.4.4 The set of environment names and the method for altering the environment list used by the getenv function

The `getenv` function always returns `NULL`.

7.10.4.5 The contents and mode of execution of the string by the system function

The `system` function always returns 0 and has no effect.

7.11.6.2 The contents of the error message strings returned by the strerror function

"There are no error strings defined!".

7.12.1 The local time zone and Daylight Saving Time

This implementation of `time.h` does not support either daylight saving or time zones and hence this function will interpret the argument as Coordinated Universal Time (UTC).

7.12.2.1 The era for the clock function

The era for the clock is the number of clock ticks since the start of program execution.

ISO/IEC 9899:1999 C Standard (C99 Mode)

The contents of this section refer to Annex J of the ISO/IEC 9899:1999 C Standard; the subsection numbers refer to parts of that Standard which have implementation-defined aspects.

J3.1 Translation

3.10, 5.1.1.3 How a diagnostic is identified

The compiler will emit descriptive diagnostics via the standard error stream at compilation time (e.g. "cc0223: function declared implicitly") or as annotations in generated assembly files.

5.1.1.2 Whether each non-empty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3

Non-empty sequences of white-space characters are retained in translation phase 3.

J3.2 Environment

5.1.1.2 The mapping between physical source file multi-byte characters and the source character set in translation phase 1

When a multi-byte character is encountered, the compiler will interpret the constituent bytes as ASCII characters irrespective of what was intended by the author.

5.1.2.1 The name and type of the function called at program startup in a freestanding environment

The name of the function called at program startup is:

```
int main();
```

or, alternatively:

```
int main(int argc, char *argv[]);
```

5.1.2.1 The effect of program termination in a freestanding environment

On program termination, functions registered by the `atexit` function are called in reverse order of registration and then the processor is placed in an `IDLE` state.

5.1.2.2.1 An alternative manner in which the main function may be defined

The default startup code source, which calls '`main`', is provided and can be configured by the user.

Alternatively, startup code can be generated in the project settings within the IDE.

5.1.2.2.1 The values given to the strings pointed to by the argv argument to main

By default, `argv[0]` is a `NULL` pointer.

The values given to the strings pointed to by the `argv` argument can be defined by the user. For more information, see Support for argv/argc.

5.1.2.3 What constitutes an interactive device

An interactive device is considered a paired display screen and keyboard.

7.14 The set of signals, their semantics, and their default handling

The following signals are supported:

- `SIGTERM`
- `SIGABRT`
- `SIGFPE`
- `SIGILL`
- `SIGINT`

- `SIGSEGV`

7.14 After the handler is invoked, the disposition of the signal is not reset to SIG_DFL

By default, these signals are ignored.

7.14.1.1 Signal values other than SIGFPE, SIGILL, and SIGSEGV that correspond to a computational exception

There are no other signal values that correspond to a computational exception.

7.14.1.1 Signals for which the equivalent of signal(sig, SIG_IGN); is executed at program startup

- `SIGTERM`
- `SIGABRT`
- `SIGFPE`
- `SIGILL`
- `SIGINT`
- `SIGSEGV`

7.20.4.5 The set of environment names and the method for altering the environment list used by the getenv function

There is no default operating system and `getenv` will always return `NULL`.

7.20.4.6 The manner of execution of the string by the system function

The `system` function always returns 0.

J3.3 Identifiers

6.4.2 Which additional multi-byte characters may appear in identifiers and their correspondence to universal character names

Multi-byte characters may not be used in identifiers.

5.2.4.1, 6.4.2 The number of significant initial characters in an identifier

The maximum number of significant initial characters in an identifier is 15,000.

J3.4 Characters

The number of bits in a byte

When the `-char-size-8` switch is used on processors with support for byte-addressing (see the ***Processors Supporting Byte-Addressing*** table in Processor Features), a byte is 8 bits in size. Otherwise it is 32 bits in size.

5.2.1 The values of the members of the execution character set

The values of the execution character set are shown in the ***The Execution Character Set*** table (with unprintable characters left blank).

Table 2-65: The Execution Character Set

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0																
0x1																
0x2	(space)	!	"	#	$	%	&	`	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	(DEL)

5.2.2 The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences

These values are shown in the ***Escape Sequences in the Execution Character Set*** table.

Table 2-66: Escape Sequences in the Execution Character Set

Escape	*Value*
\a	0x7
\b	0x8
\f	0xC
\n	0xA
\r	0xD
\t	0x9
\v	0xB

6.2.5 The value of a char object into which has been stored any character other than a member of the basic execution character set

The resulting value is derived from a cast of the character to `signed int`, which will not affect the value of the character.

6.2.5, 6.3.1.1 Which of signed char or unsigned char has the same range, representation and behavior as "plain" char

A "plain" `char` has the same range and value as a `signed char`.

6.4.4.4, 5.1.1.2 The mapping of members of the source character set (in character constants and string literals) to members of the execution character set

Characters in the source file are interpreted as ASCII values, which are the same values used in the execution environment.

6.4.4.4 The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character

An integer character constant may contain 1 character. Using more than 1 character will result in warning cc2226 being issued and all but the last character being discarded. No escape characters other than those specified in the C99 standard are supported, and these all map to a single byte in the execution environment.

6.4.4.4 The value of a wide character constant containing more than one multi-byte character, or containing a multi-byte character or escape sequence not represented in the extended execution character set

Where a wide character contains more than one multi-byte character, only the first character is retained and warning cc0026 will be issued. Subsequent characters are discarded. No escape characters other than those specified in the C99 standard are supported, and these all map to a single byte in the execution environment.

6.4.4.4 The current locale used to convert a wide character constant consisting of a single multi-byte character that maps to a member of the extended execution character set into a corresponding wide character code

Only the "C" locale is supported in Analog Devices toolchain and processors.

6.4.5 The current locale used to convert a wide string literal into corresponding wide character codes

Only the "C" locale is supported in Analog Devices toolchain and processors.

6.4.5 The value of a string literal containing a multi-byte character or escape sequence not represented in the execution character set

There are no escape sequences outside the basic or extended character sets.

J3.5 Integers

6.2.5 Any extended integer types that exist in the implementation

None.

6.2.6.2 Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value

Two's Complement:

- The sign bit being 1 and all value bits being zero is considered a normal number.

6.3.1.1 The rank of any extended integer type relative to another extended integer type with the same precision

N/A.

6.3.1.3 The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type

The hexadecimal value is copied and then interpreted as signed. e.g. `MAX_UINT` becomes -1.

6.5 The results of some bitwise operations on signed integers

Right shifts will retain the sign bit on a signed integer. All other bitwise operations treat signed integers as unsigned.

J3.6 Floating-Point

5.2.4.2.2 The accuracy of the floating-point operations and of the library functions in the <math.h> and <complex.h> that return floating-point results.

This is a conforming freestanding implementation of C99. The accuracy of the library functions in these headers are therefore undocumented.

5.2.4.2.2 The rounding behaviors characterized by non-standard values of FLT_ROUNDS

`FLT_ROUNDS` is a standard value.

5.2.4.2.2 The evaluation methods characterized by non-standard negative values of FLT_EVAL_METHOD

`FLT_EVAL_METHOD` is undefined.

6.3.1.4 The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value

Round to nearest, ties to even.

6.3.1.5 The direction of rounding when a floating-point number is converted to a narrower floating-point number

Round to nearest, ties to even.

6.4.4.2 How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants

`FLT_RADIX` is defined as 2 in `<float.h>`, so floating-point constants are represented using standards-conforming rounding.

6.5 Whether and how floating expressions are contracted when not disallowed by the FP_CONTRACT pragma

This is a conforming freestanding implementation of C99. The `FP_CONTRACT` pragma is therefore not supported.

7.6.1 The default the state for the FENV_ACCESS pragma

This is a conforming freestanding implementation of C99, and the `FENV_ACCESS` pragma is only used for accessing the floating-point environment `fenv.h` - a header not required for such an implementation. As such The pragma is not supported.

7.6, 7.12 Additional floating-point exceptions, rounding modes, environments and classification, and their macro names

There are no additional floating-point exceptions, rounding modes, environments or classifications.

7.12.2 The default state for the FP_CONTRACT pragma

This is a conforming freestanding implementation of C99. The `FP_CONTRACT` pragma is therefore not supported.

F.9 Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation

The "inexact" floating-point exception is not supported for SHARC processors.

F.9 Whether the "underflow" (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation

The "inexact" floating-point exception is not supported on SHARC processors.

The "underflow" floating-point exception is not enabled by default on SHARC processors.

ISO/IEC 14822:2003 C++ Standard (C++ Mode)

The subsection of this section refer to parts of the ISO/IEC 14822:2003 C++ Standard which have implementation-defined aspects.

1.7 The C++ Memory Model

The fundamental storage unit in the C + + memory model is the byte. A byte is at least large enough to contain any member of the basic execution character set and is composed of a contiguous sequence of bits, the number of which is implementation-defined.

When the `-char-size-8` switch is used on processors with support for byte-addressing (see the ***Processors Supporting Byte-Addressing*** table in Processor Features), a byte is 8 bits in size. Otherwise it is 32 bits in size.

1.9 Program Execution

What constitutes an interactive device is implementation-defined.

An interactive device is considered a paired display screen and keyboard.

2.1 Phases of Translation

Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set (introducing new-line characters for end-of-line indicators) if necessary.

Characters in the source file are interpreted as ASCII values, which are also used in the execution environment.

Whether each non-empty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

Non-empty sequences of white-space characters are retained.

It is implementation-defined whether the source of the translation units containing these definitions is required to be available.

The source of the translation units containing these definitions must be available.

2.2 Character Sets

The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.

The values of the execution character set are shown in the ***Execution Character Set for C++ Mode*** table (with unprintable characters left blank).

Table 2-67: Execution Character Set for C++ Mode

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0																
0x1																
0x2	(space)	!	"	#	$	%	&	`	(	)	*	+	,	-	.	/
0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	(DEL)

2.13.2 Character Literals

A multi-character literal has type int and implementation-defined value.

An integer character constant may contain 1 character. If more than 1 character is used, error cc0026 is issued.

The value of a wide-character literal containing multiple c-chars is implementation-defined.

Where a wide character contains more than one multi-byte character, only the first character is retained and warning cc0026 will be issued. Subsequent characters are discarded.

The value of a character literal is implementation-defined if it falls outside of the implementation-defined range defined for char (for ordinary literals) or wchar_t (for wide literals).

The full 32-bit value is retained.

2.13.4 String Literals

Whether all string literals are distinct (that is, are stored in non-overlapping objects) is implementation-defined.

Identical string literals within the same object file will not be distinct. That is, only one copy of the string will exist.

3.6.1 Main Function

An implementation shall not predefine the main function. This function shall not be overloaded. It shall have a return type of type int, but otherwise its type is implementation-defined.

The name of the function called at program startup is:

```
int main();
```

or, alternatively:

```
int main(int argc, char *argv[]);
```

The linkage (3.5) of main is implementation-defined.

`main` has external "C" linkage.

3.6.2 Initialization of Non-Local Objects

It is implementation-defined whether or not the dynamic initialization (8.5, 9.4, 12.1, 12.6.1) of an object of namespace scope is done before the first statement of main.

Dynamic initialization of an object of namespace scope is done before the first statement of `main`.

3.9 Types

For POD types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.

All POD types are represented in the same format as in C.

The alignment of a complete object type is an implementation-defined integer value representing a number of bytes; an object is allocated at an address that meets the alignment requirements of its object type.

Compound types (structs, classes) are aligned on the boundary that matches the alignment of the most strictly-aligned member of the type.

Top-level global arrays are always aligned on a double-word boundary, regardless of the underlying type. Arrays within structures are not aligned beyond the required alignment for their type.

3.9.1 Fundamental Types

It is implementation-defined whether a char object can hold negative values.

A `char` can hold negative values.

The value representation of floating-point types is implementation-defined.

The representations of the floating point types are as follows:

- `float`
  - 32 bits (1 sign bit, 8 exponent bits, 32 mantissa bits)

    -3.4028234663852886E+38 to 3.4028234663852886E+38
- `double` (default setting)
  - 32 bits (1 sign bit, 8 exponent bits, 32 mantissa bits)

    -3.4028234663852886E+38 to 3.4028234663852886E+38
- `double` (when compiling with `-double-size-64`)
  - 64 bits (1 sign bit, 11 exponent bits, 52 mantissa bits)

    -1.797693134862315708e+308 to 1.797693134862315708e+308

- `long double`
  - 64 bits (1 sign bit, 11 exponent bits, 52 mantissa bits)

    -1.797693134862315708e+308 to 1.797693134862315708e+308

3.9.2 Compound Types

The value representation of pointer types is implementation-defined.

Pointer types are represented as 32-bit unsigned integers.

4.7 Integral Conversions

If the destination type is signed, the value is unchanged if it can be represented in the destination type (and bit-field width); otherwise, the value is implementation-defined.

When converting a signed integer to a smaller signed integer, the lower bits of the signed integer (of the size of the smaller signed integer) are copied to the smaller signed integer. If the topmost copied bit is set, this will result in a negative number.

4.8 Floating-Point Conversions

If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values.

Round to nearest, ties to even.

4.9 Floating-Integral Conversions

An rvalue of an integer type or of an enumeration type can be converted to an rvalue of a floating-point type. The result is exact if possible. Otherwise, it is an implementation-defined choice of either the next lower or higher representable value.

Round to nearest, ties to even.

5.2.8 Type Identification

The result of a typeid expression is an lvalue of static type const std::type_info (18.5.1) and dynamic type const std::type_info or const name where name is an implementation-defined class derived from std::type_info which preserves the behavior described in 18.5.1.

The result of a `typeid` expression is an lvalue of static type `const std::type_info` and dynamic type `const std::type_info`.

5.2.10 Reinterpret Cast

The mapping performed by reinterpret_cast is implementation-defined.

For an expression "`reinterpret_cast<T>(v)`", the bits in the object representation of "`v`" will be treated as type as an object of type "`T`".

A pointer can be explicitly converted to any integral type large enough to hold it. The mapping function is implementation-defined.

The bit pattern of the pointer is interpreted as the integral type. No sign extension is performed if the integral type is larger than the pointer.

A value of integral type or enumeration type can be explicitly converted to a pointer. A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation-defined.

A cast from pointer to integer results in the most-significant bits being discarded if the size of the pointer is larger than the integer. If the pointer is smaller than the integer type being cast to, the integer will be zero-extended.

A cast from integer to pointer results in the most-significant bits being discarded if the size of the integer is larger than the pointer. If the integer is smaller than the pointer type being cast to, the pointer will be sign-extended.

5.3.3 Sizeof

sizeof(char), sizeof(signed char) and sizeof(unsigned char) are 1; the result of sizeof applied to any other fundamental type (3.9.1) is implementation-defined. [Note: in particular, sizeof(bool) and sizeof(wchar_t) are implementation-defined.]

Sizes are as shown in the ***Sizes of C++ Standard Types*** table.

Table 2-68: Sizes of C++ Standard Types

Type	*Sizeof with -char-size-32*	*Sizeof with -char-size-8*
`char` (signed, unsigned)	1	1
`short` (signed, unsigned)	1	2
`int` (signed, unsigned)	1	4
`long` (signed, unsigned)	1	4
`long long` (signed, unsigned)	2	8
`float`	1	4
`double` (default)	1	4
`double` (`-double-size-64`)	2	8
`long double`	2	8
`bool`	1	1
`wchar_t`	1	4

5.6 Multiplicative Operators

The binary / operator yields the quotient, and the binary % operator yields the remainder from the division of the first expression by the second. If the second operand of / or % is zero the behavior is undefined; otherwise (a/b)*b + a

%b is equal to a. If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined.

If the first operand is negative, the sign of the remainder will be negative, otherwise the sign of the remainder is nonnegative.

5.7 Additive Operators

When two pointers to elements of the same array object are subtracted, the result is the difference of the subscripts of the two array elements. The type of the result is an implementation-defined signed integral type; this type shall be the same type that is defined as ptrdiff_t in the <cstddef> header (18.1).

The type of the result is `int`.

5.8 Shift Operators

The value of E1 >> E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of E1 divided by the quantity 2 raised to the power E2. If E1 has a signed type and a negative value, the resulting value is implementation-defined.

Right shifts will retain the sign bit on a signed integer.

7.1.5.2 Simply Type Specifiers

It is implementation-defined whether bit-fields and objects of char type are represented as signed or unsigned quantities.

By default, bit-fields and objects of `char` type are represented as signed quantities.

Bit-fields can represented as unsigned quantities by using the compiler switch `-unsigned-bitfield`.

7.2 Enumeration Declarations

It is implementation-defined which integral type is used as the underlying type for an enumeration except that the underlying type shall not be larger than int unless the value of an enumerator cannot fit in an int or unsigned int.

The underlying type for an enumeration shall be `int`.

7.4 The asm Declaration

The meaning of an asm declaration is implementation-defined.

For more information, see Inline Assembly Language Support Keyword (asm).

7.5 Linkage Specifications

The string-literal indicates the required language linkage. The meaning of the string-literal is implementation-defined.

Three string-literals are supported:

- `"C"` - the function name in the source file is prefixed with an underscore ("_") in the object file.

- `"C++"` - the function name is mangled according to the compiler's name mangling rules.
- `"asm"` - the function name in the source file is used in the object file without a prefix or name-mangling.

Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation-defined and language-dependent.

Three string-literals are supported:

- `"C"` - the function name in the source file is prefixed with an underscore ("_") in the object file.
- `"C++"` - the function name is mangled according to the compiler's name mangling rules.
- `"asm"` - the function name in the source file is used in the object file without a prefix or name-mangling.

9.6 Bit-Fields

Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined.

Bit-fields are stored using a big-endian representation on processors without support for byte-addressing, or when the `-char-size-32` switch is used. Bit-fields are stored using a little-endian representation when the `-char-size-8` switch is used on processors with support for byte-addressing.

Bit-fields are aligned such that they do not cross a 32-bit word boundary (for bit-fields of type `char`, `short`, `int` or `long`) or a 64-bit boundary (for bit-fields of type `long long`). For example, a 24-bit bit-field can be placed immediately after an 8-bit bit-field, but a 25-bit bit-field member will be aligned on the next 32-bit boundary.

It is implementation-defined whether a plain (neither explicitly signed nor unsigned) char, short, int or long bit-field is signed or unsigned.

Plain bit-fields are signed.

14 Templates

A template name has linkage (3.5). A non-member function template can have internal linkage; any other template name shall have external linkage. Entities generated from a template with internal linkage are distinct from all entities generated in other translation units. A template, a template explicit specialization (14.7.3), or a class template partial specialization shall not have C linkage. If the linkage of one of these is something other than C or C++, the behavior is implementation-defined.

Only C++ linkage is supported for templates.

14.7.1 Implicit Instantiation

There is an implementation-defined quantity that specifies the limit on the total depth of recursive instantiations, which could involve more than one template.

The limit on the total depth of recursive instantiations is 64.

15.5.1 The terminate() Function

In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before terminate() is called.

The stack is not unwound before the call to `terminate()`.

15.5.2 The unexpected() Function

If the exception-specification does not include the class std::bad_exception (18.6.2.1) then the function terminate() is called, otherwise the thrown exception is replaced by an implementation-defined object of the type std::bad_exception and the search for another handler will continue at the call of the function whose exception-specification was violated.

The object of the type `std::bad_exception` will contain the string "bad exception".

16.1 Conditional Inclusion

Whether the numeric value for these character literals matches the value obtained when an identical character literal occurs in an expression (other than within a #if or #elif directive) is implementation-defined.

The numeric value for these character literals matches the value obtained when an identical character literal occurs in an expression.

Also, whether a single-character character literal may have a negative value is implementation-defined.

A single-character may have a negative value.

16.2 Source File Inclusion

Searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

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:

- The directory containing the current input file (the primary source file or the file containing the `#include`).
- Any directories specified with the `-I` switch (`-I directory[{,|;} directory...]`) in the order they are listed on the command line.
- Any directories on the standard list: `<install_path>\...\include`.

The mapping between the delimited sequence and the external source file name is implementation-defined.

The source file character sequence is mapped to its corresponding ASCII character sequence.

A #include preprocessing directive may appear in a source file that has been read because of a #include directive in another file, up to an implementation-defined nesting limit.

The compiler does not define a nesting limit for `#include` directives.

16.6 Pragma Directive

A preprocessing directive #pragma causes the implementation to behave in an implementation-defined manner.

For more information, see Pragmas.

16.8 Predefined Macro Names

If the date of translation is not available, an implementation-defined valid date is supplied.

The macro `__DATE__` will be defined as "[date unknown]".

If the time of translation is not available, an implementation-defined valid time is supplied.

The macros `__TIME__` will be defined as "[time unknown]".

Whether __STDC__ is predefined and if so, what its value is, are implementation-defined.

`__STDC__` is predefined with the value 1.

17.4.4.5 Reentrancy

Which of the functions in the C++ Standard Library are not reentrant subroutines is implementation-defined.

The following functions are not reentrant in the C++ Standard library, as implemented in CCES:

- Functions that use streams.
- Dynamic memory allocation functions (`new`, `delete`, etc.).
- The exceptions handling support routines.

Although these functions are not reentrant, thread-safe versions of them are implemented in the multi-threaded C++ library. For more information, see ***Using the Libraries in a Multi-Threaded Environment*** in the *C/C++ Library Manual for SHARC Processors*.

17.4.4.8 Restrictions on Exception Handling

Any other functions defined in the C++ Standard Library that do not have an exception-specification may throw implementation-defined exceptions unless otherwise specified.

The ***Functions Which Throw Exceptions*** table shows which functions may throw the following exceptions, if the application is built with exceptions enabled.

Table 2-69: Functions Which Throw Exceptions

Function	*Exception Type*
ios_base::clear	failure
locale::locale	runtime_error
_Locinfo::_Addcats	runtime_error
_String_base::_Xlen	length_error

Table 2-69: Functions Which Throw Exceptions (Continued)

Function	*Exception Type*
_String_base::_Xran	out_of_range
array new and delete operators	bad_alloc

18.3 Start and Termination

Exit() - Finally, control is returned to the host environment. If status is zero or EXIT_SUCCESS, an implementation-defined form of the status successful termination is returned. If status is EXIT_FAILURE, an implementation-defined form of the status unsuccessful termination is returned. Otherwise the status returned is implementation-defined.

The program will idle at the label `__lib_prog_term`. R0 (the return register) will contain the status.

18.4.2.1 Class bad_alloc

The result of calling what() on the newly constructed object is implementation-defined.

`what()` will return the string "bad allocation".

virtual const char* what() const throw(); Returns: An implementation-defined NTBS.

`what()` will return the string "bad allocation".

18.5.1 Class type_info

const char* name() const; Returns: an implementation-defined NTBS.

The ***Strings returned by name()*** table shows the string returned by the name() function for the basic types.

Table 2-70: Strings returned by name()

Type	*String*
`bool`	`b`
`char`	`c`
`signed char`	`a`
`unsigned char`	`h`
(signed) `short`	`s`
`unsigned short`	`t`
(signed) `int`	`i`
`unsigned int`	`j`
(signed) `long`	`l`
`unsigned long`	`m`
(signed) `long long`	`x`

Table 2-70: Strings returned by name() (Continued)

Type	*String*
`unsigned long long`	`y`
`float`	`f`
`double`	`d`
`long double`	`e`
`wchar_t`	`w`

18.5.2 Class bad_cast

virtual const char* what() const throw(); Returns: An implementation-defined NTBS.

Calling `what()` will return the string "bad cast".

18.5.3 Class bad_typeid

bad_typeid() throw(); Notes: The result of calling what() on the newly constructed object is implementation-defined.

Calling `what()` will return the string "bad typeid".

virtual const char* what() const throw(); Returns: An implementation-defined NTBS.

Calling `what()` will return the string "bad typeid".

18.6.1 Class Exception

exception& operator=(const exception&) throw(); Notes: The effects of calling what() after assignment are implementation-defined.

Calling `what()` will return the string "unknown".

virtual const char* what() const throw(); Returns: An implementation-defined NTBS.

Calling `what()` will return the string "unknown".

18.6.2.1 Class bad_exception

bad_exception() throw(); Notes: The result of calling what()on the newly constructed object is implementation-defined.

Calling `what()` will return the string "bad exception".

virtual const char* what() const throw(); Returns: An implementation-defined NTBS.

Calling `what()` will return the string "bad exception".

21 Strings Library

The type streampos is an implementation-defined type that satisfies the requirements for POS_T in 21.1.2.

`streampos` is a `typedef` of the `fpos` class.

The type streamoff is an implementation-defined type that satisfies the requirements for OFF_T in 21.1.2.

`streamoff` is a `typedef` of the `long` type.

The type mbstate_t is defined in <cwchar> and can represent any of the conversion states possible to occur in an implementation-defined set of supported multi-byte character encoding rules.

Multi-byte characters are not supported in Analog Devices Compiler, so no multi-byte characters may be used in identifiers.

21.1.3.2 struct char_traits<wchar_t>

The type wstreampos is an implementation-defined type that satisfies the requirements for POS_T in 21.1.2.

The type `wstreampos` not supported in Analog Devices toolset.

The type mbstate_t is defined in <cwchar> and can represent any of the conversion states possible to occur in an implementation-defined set of supported multi-byte character encoding rules.

Multi-byte characters are not supported in Analog Devices Compiler, so no multi-byte characters may be used in identifiers.

22.1.1.3 Locale Members

*basic_string<char> name() const; Returns: The name of *this, if it has one; otherwise, the string "*". If *this has a name, then locale(name().c_str()) is equivalent to *this. Details of the contents of the resulting string are otherwise implementation-defined.*

`name` returns the name of `*this`, if it has one; otherwise, the string "*".

22.2.1.3 ctype Specializations

The implementation-defined value of member table_size is at least 256.

The value of member `table_size` is 256.

22.2.1.3.2 ctype<char> Members

In the following member descriptions, for unsigned char values v where (v >= table_size), table()[v] is assumed to have an implementation-defined value (possibly different for each such value v) without performing the array look-up.

As `table_size` has the value 256, it is not possible for `v` to be greater than or equal to `table_size`.

22.2.5.1.2 time_get Virtual Functions

iter_type do_get_year(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm t) const; Effects: Reads characters starting at s until it has extracted an unambiguous year identifier. It is implementation-defined whether two-digit year numbers are accepted, and (if so) what century they are assumed to lie in. Sets the t->tm_yearmember accordingly.*

If the two-digit year is less than '69', it is assumed that the year is in the 21st century (i.e. 2000 -> 2068); otherwise, it is assumed that the year is in the 20th century.

22.2.5.3.2 time_put Virtual Functions

Effects: Formats the contents of the parameter t into characters placed on the output sequence s. Formatting is controlled by the parameters format and modifier, interpreted identically as the format specifiers in the string argument to the standard library function strftime(). except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.

The ***Outputs for time_put Specifiers*** table shows the character sequences produced for each specifier that depends on the C locale.

Table 2-71: Outputs for time_put Specifiers

Specifier	*Characters*
%a	"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"
%A	"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
%b	"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
%B	"January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"
%c	Date and time in the format: DDD MMM DD HH:MM:SS YYYY For example, "Sat Jan 31 23:59:59 2011".
%p	"AM", "PM"
%x	Date in the format: MM/DD/YY For example, "12/31/12".
%X	Time in the format: HH:MM:SS For example, "23:59:59".

22.2.7.1.2 Messages Virtual Functions

catalog do_open(const basic_string<char>& name, const locale& loc) const; Returns: A value that may be passed to get()to retrieve a message, from the message catalog identified by the string name according to an implementation-defined mapping. The result can be used until it is passed to close().

This function has no effect.

string_type do_get(catalog cat, int set, int msgid, const string_type& dfault) const; Returns: A message identified by arguments set, msgid, and dfault, according to an implementation-defined mapping.

The function `do_get` always returns the string pointed to by `dfault`.

void do_close(catalog cat) const; Notes: The limit on such resources, if any, is implementation-defined.

This function has no effect.

26.2.8 Complex Transcendentals

The value returned for pow(0,0) is implementation-defined.

This is a conforming freestanding implementation of C++. Complex transcendentals are not supported.

27.1.2 Positioning Type Limitations

The classes of clause 27 with template arguments charT and traits behave as described if traits::pos_type and traits::off_type are streampos and streamoff respectively. Except as noted explicitly below, their behavior when traits::pos_type and traits::off_type are other types is implementation-defined.

`traits::pos_type` and `traits::off_type` are `streampos` and `streamoff`, respectively.

27.4.1 Types

The type streamoff is an implementation-defined type that satisfies the requirements of 27.4.3.2.

`streamoff` is of type `long`.

27.4.2.4 ios_base Static Members

bool sync_with_stdio(bool sync = true); Effects: If any input or output operation has occurred using the standard streams prior to the call, the effect is implementation-defined.

`iostream` objects are always synchronised with the standard streams. This function has no effect.

27.4.4.3 basic_ios iostate Flags Functions

If (rdstate() & exceptions()) == 0, returns. Otherwise, the function throws an object fail of class basic_ios::failure (27.4.2.1.1), constructed with implementation-defined argument values.

If '`ios_base::badbit`' is set, the exception is created with the string `"ios_base::badbit set"`.

If '`ios_base::failbit`' is set, the exception is created with the string `"ios_base::failbit set"`.

27.7.1.3 Overridden Virtual Functions

basic_streambuf<charT,traits> setbuf(charT* s, streamsize n); Effects: implementation-defined, except that set-buf(0,0)has no effect.*

`streambuf()` has no effect.

27.8.1.4 Overridden Virtual Functions

basic_streambuf setbuf(char_type* s, streamsize n); Effects: If setbuf(0,0) is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are implementation-defined.*

If `setbuf(s, n)` is called before any I/O has occurred, the buffer '`s`', of size '`n`', is used by the I/O routines. Calls to `setbuf()` on a stream after I/O has occurred are ignored.

int sync(); Effects: If a put area exists, calls filebuf::overflow to write the characters to the file. If a get area exists, the effect is implementation-defined.

The `sync()` function has no effect on the get area.

C.2.2.3 Macro NULL

The macro NULL, defined in any of <clocale>, <cstddef>, <cstdio>, <cstdlib>, <cstring>, <ctime>, or <cwchar>, is an implementation-defined C++ null pointer constant in this International Standard (18.1).

The macro `NULL` is defined as 0.

D.6 Old iostreams Members

The type streamoff is an implementation-defined type that satisfies the requirements of type OFF_T (27.4.1).

`streamoff` is a `typedef` of the '`long`' type.

The type streampos is an implementation-defined type that satisfies the requirements of type POS_T (27.2).

`streampos` is a `typedef` of the `fpos` class.

3 Optimal Performance from C/C++ Source Code

This chapter provides guidance on tuning your application to achieve the best possible code from the compiler. Since implementation choices are available when coding an algorithm, understanding their impact is crucial to attaining optimal performance.

This chapter contains:

- General Guidelines
- Improving Conditional Code
- Loop Guidelines
- Using Built-In Functions in Code Optimization
- Smaller Applications: Optimizing for Code Size
- Using Pragmas for Optimization
- Useful Optimization Switches
- How Loop Optimization Works
- Assembly Optimizer Annotations
- Analyzing Your Application

This chapter helps you get maximal code performance from the compiler. Most of these guidelines also apply when optimizing for minimum code size, although some techniques specific to that goal are also discussed.

The first section looks at some general principles, and explains how the compiler can help your optimization effort. Optimal coding styles are then considered in detail. Special features such as compiler switches, built-in functions, and pragmas are also discussed. The chapter includes a short example to demonstrate how the optimizer works.

Small examples are included throughout this chapter to demonstrate points being made. Some show recommended coding styles, while others identify styles to be avoided or code that it may be possible to improve. These are commented in the code as "GOOD" and "BAD" respectively.

General Guidelines

This section contains:

- How the Compiler Can Help
- The volatile Type Qualifier
- Data Types
- Getting the Most From IPA
- Indexed Arrays Versus Pointers
- Using Function Inlining
- Using Inline asm Statements
- Memory Usage

Remember the following strategy when writing an application:

1. Choose the language as appropriate. Your first decision is whether to implement your application in C or C++. Performance considerations may influence this decision. C++ code using only C features has very similar performance to pure C code. Many higher level C++ features (for example, those resolved at compilation, such as namespaces, overloaded functions and also inheritance) have no performance cost.

   However, use of some other features may degrade performance. Carefully weigh performance loss against the richness of expression available in C++ (such as virtual functions or classes used to implement basic data types).

2. Choose an algorithm suited to the architecture being targeted. For example, the target architecture will influence any trade-off between memory usage and algorithm complexity.

3. Code the algorithm in a simple, high-level generic form. Keep the target in mind, especially when choosing data types.

4. Tune critical code sections. After your application is complete, identify the most critical sections. Carefully consider the strengths of the target processor and make non-portable changes where necessary to improve performance.

How the Compiler Can Help

The compiler provides many facilities to help the programmer to achieve optimal performance, including the compiler optimizer, Profile-Guided Optimizer (PGO), and interprocedural optimizers.

This section contains:

- Using the Compiler Optimizer
- Using Compiler Diagnostics
- Using Profile-Guided Optimization

- Using Interprocedural Optimization

Using the Compiler Optimizer

There is a vast difference in performance between code compiled optimized and code compiled non-optimized. In some cases, optimized code can run ten or twenty times faster. Always use optimization when measuring performance or shipping code as product.

The optimizer in the C/C++ compiler is designed to generate efficient code from source that has been written in a straightforward manner. The basic strategy for tuning a program is to present the algorithm in a way that gives the optimizer the best possible visibility of the operations and data, and hence the greatest freedom to safely manipulate the code. Future releases of the compiler will continue to enhance the optimizer. Expressing algorithms simply will provide the best chance of benefiting from such enhancements.

Note that the default setting (or "debug" mode within the IDE) is for non-optimized compilation in order to assist programmers in diagnosing problems with their initial coding. The optimizer is enabled through the IDE by selecting ***Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > General > Enable optimization***, or by using the `-O` switch (`-O[0|1]`). A "release" build from within the IDE automatically enables optimization.

Using Compiler Diagnostics

There are many features of the C and C++ languages that, while legal, often indicate programming errors. There are also aspects that are valid but may be relatively expensive for an embedded environment. The compiler can provide diagnostics which save time and effort in characterizing source-related problems:

These diagnostics are particularly important for obtaining high-performance code, since the optimizer aggressively transforms the application to get the best performance, discarding unused or redundant code; if this code is redundant because of a programming error (such as omitting an essential `volatile` qualifier from a declaration), then the code will behave differently from a non-optimized version. Using the compiler's diagnostics may help you identify such situations before they become problems.

The diagnostic facilities are described in the following sections:

- Warnings, Annotations and Remarks
- Run-Time Diagnostics
- Steps for Developing Your Application

Warnings, Annotations and Remarks

By default, the compiler emits warnings to the standard error stream at compile-time when it detects a problem with the source code. Warnings can be disabled individually, with the `-Wsuppress` switch (`-W{annotation|error|remark|suppress|warn}` *`number[, number ...]`*) or as a class, with the `-w` switch (`-w`), disabling all warnings and remarks. However, disabling warnings is inadvisable until each instance has been investigated for problems.

A typical warning would be: a variable being used before its value has been set.

Remarks are diagnostics that are less severe than warnings. Like warnings, they are produced at compile-time to the standard error stream, but unlike warnings, remarks are suppressed by default. Remarks are typically for situations that are probably correct, but less than ideal. Remarks may be enabled as a class with the `-Wremarks` switch (`-Wremarks`) or by setting ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Warning*** > ***Warning/annotation/remark control*** to ***Errors, warnings, annotations and remarks*** in the IDE.

A typical remark would be: a variable being declared, but never used.

A remark may be promoted to a warning through the `-Wwarn` switch (`-W{annotation|error|remark|suppress|warn} number[, number ...]`). Remarks and warnings may be promoted to errors through the `-Werror` switch (`-W{annotation|error|remark|suppress|warn} number[, number ...]`).

Annotations are diagnostics that are between warnings and remarks in severity. Like remarks, annotations are usually suppressed. Where remarks comment on the input source file, annotations provide information about the code the compiler has generated from the source file.

A typical annotation would be: using a volatile variable within a loop limits optimization.

Both annotations and remarks can be viewed in the IDE; they are listed as "infos" in the ***Problems*** view, and an "information" icon appears in the gutter of the source file's view, adjacent to the associated line. Hovering over the gutter icon displays the annotations and remarks for the line.

Annotations are also emitted to the generated assembly file, as comments. For more information, see Assembly Optimizer Annotations.

Run-Time Diagnostics

Although the compiler can identify many potential problems through its static analysis, some problems only become apparent at runtime. The compiler and libraries provide a number of facilities for assisting in identifying such problems. These facilities are:

- Run-time diagnostics, where the compiler plants additional code to check for common programming errors. For more information, see Run-Time Checking.
- Stack overflow detection, where the compiler checks that the stack does not run out of space. For more information, see Stack Overflow Detection.
- Heap debugging, where the compiler links the application with an enhanced version of the heap library, to detect memory leaks and other common dynamic-memory issues. For more information, see Heap Debugging.

Steps for Developing Your Application

To improve overall code quality:

1. Enable remarks and build the application. Gather all warnings and remarks generated.
2. Examine the generated diagnostics, and choose those message types that you consider most important. For example, you might select just `cc0223`, a remark that identifies implicitly-declared functions.

3. Promote those remarks and warnings to errors, using the `-Werror` switch (for example, "`-Werror 0223`"), and rebuild the application. The compiler will now fault such cases as errors, so you will have to fix the source to address the issues before your application will build.
4. Once your application rebuilds, repeat the process for the next most important diagnostics.
5. When you have dealt with the diagnostics you consider significant, rebuild your application with run-time diagnostics enabled, and run your regression tests, to see whether any problems lurk. (Given the overheads of run-time diagnostics, you will probably find it better to only enable one form at a time.)
6. Once your application runs successfully with each form of run-time diagnostic, disable run-time diagnostics and rebuild your application for release.

Diagnostics you might typically consider first include:

- `cc0223`: function declared implicitly
- `cc0549`: variable used before its value is set
- `cc1665`: variable is possibly used before its value is set, in a loop
- `cc0187`: use of "=" where "==" may have been intended
- `cc1045`: missing return statement at the end of non-void function
- `cc0111`: statement is unreachable

If you have particular cases that are correct for your application, do not let them prevent your application from building because you have raised the diagnostic to an error. For such cases, temporarily lower the severity again within the source file in question by using `#pragma diag`. See Modifying the Behavior of an Entire Class of Diagnostics.

Using Profile-Guided Optimization

Profile-guided optimization (PGO) is an excellent way to tune the compiler's optimization strategy for the typical run-time behavior of a program. There are many program characteristics that cannot be known statically at compile-time but can be provided through PGO. The compiler can use this knowledge to improve its code generation. The benefits include more accurate branch prediction, improved loop transformations, and reduced code size. The technique is most relevant where the behavior of the application over different data sets is expected to be very similar.

NOTE: The data gathered during the profile-guided optimization process can also be used to generate a code coverage report. See Profile-Guided Optimization and Code Coverage.

Profile-guided optimization can be performed on applications running on both hardware and simulators. The functionality supported and the steps required are different in each case. A summary of these differences is listed in the ***Differences Between Profile-Guided Optimization for Simulators and Hardware*** table.

Table 3-1: Differences Between Profile-Guided Optimization for Simulators and Hardware

Profile-Guided Optimization for Simulators	*Profile-Guided Optimization for Hardware*
Is non-intrusive to the application. No code or data space needs to be reserved for the profiling.	Is intrusive. Profiling requires both code and data space to be reserved in the application.
Does not impact performance. Profiling is performed in the background by the simulator.	Impacts performance. Profiling is performed on the processor as part of the application.
Does not support multi-threaded applications.	Supports multi-threaded applications.
Can only profile application where are peripherals are simulated by the simulator.	Run on hardware allowing the profiling of applications that use custom hardware.

Profile-guided optimization using the simulator is a non-intrusive process: the application code is not modified to gather the profiling data. Multi-threaded applications cannot be profiled using the simulator-based method of profile gathering.

Profile-guided optimization for applications running on hardware offers support for multi-threaded applications and applications that cannot be run on the simulator (for example, due to custom hardware or requiring input from peripherals not supported by the simulator). However, the hardware-based profiling method is more intrusive to the application, as it requires some instruction and data memory.

Using Profile-Guided Optimization With a Simulator

The process of PGO execution with a simulator is illustrated in the ***PGO Process When Targeting a Simulator*** figure.

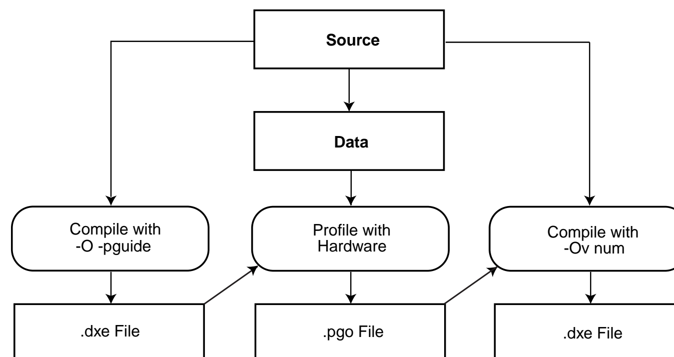

Figure 3-1: PGO Process When Targeting a Simulator

1. Compile the application with the `-pguide` switch or the ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Profile-guided Optimization*** > ***Prepare application to create new profile*** option `-pguide`. This creates an executable file containing the necessary instrumentation for gathering profile data. For best results, click ***General*** (under ***Compiler***) in the tree control and select ***Enable optimization***/`-O` switch (`-O[0|1]`) or select ***Interprocedural optimization***/`-ipa` (`-ipa`) switch.
2. Gather the profile. Run the executable under the simulator, with one or more training data sets.
   a. Load the application into the simulator.

b. Enable profiling via ***Target*** > ***PGO*** > ***Simulator*** > ***Start***.

c. Run the application with the desired training set.

d. Save the profile via ***Target*** > ***PGO*** > ***Simulator*** > ***Stop and save***.

e. Repeat the process with the next training set.

The training data sets should be representative of the data that you expect the application to process in the field. Note that unrepresentative training data sets can cause performance degradations when the application is used on real data. The profile is stored in a file with the extension `.pgo`.

3. Recompile the application using this gathered profile data:

   a. Turn off the `-pguide` switch or choose ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Profile-guided Optimization*** > ***Prepare application to create new profile*** option.

   b. Place the `.pgo` file on the command line or include it in the list of profiles under Optimize using existing profiles.

   c. Ensure optimization is enabled:

      Click ***General*** (under ***Compiler***) in the tree control and select ***Enable optimization*** option/`-O` switch (`-O[0|1]`) and/or ***Interprocedural optimization option/***`-ipa` (`-ipa`) switch.

NOTE: When C/C++ source files are specified on a compiler command line, any specified `.pgo` files will be used to guide compilation. However, any recompilation due to `.doj` files provided on the command line will reread the same `.pgo` file as when the source was previously compiled. For example, `prof2.pgo` is ignored in the following commands:

```
cc21k -O f2.c -o f2.doj prof1.pgo
cc21k -o prog.dxe f1.asm f2.doj prof2.pgo
```

An example application that demonstrates how to use PGO is in the ***PGO Process When Targeting a Simulator*** figure.

Using Profile-Guided Optimization With Hardware

The process of PGO execution with hardware is illustrated in the ***PGO on Hardware Process*** figure.

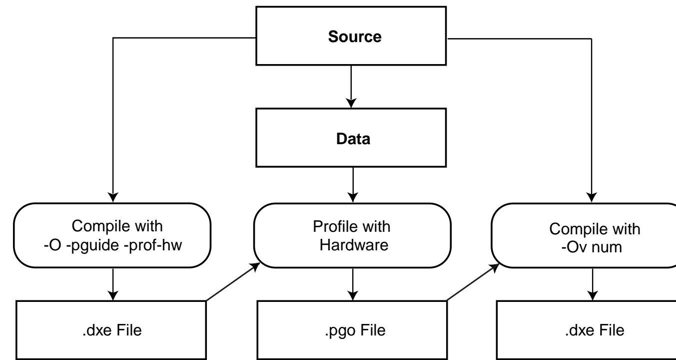

Figure 3-2: PGO on Hardware Process

1. Compile the application with:

   a. The `-pguide` switch or choose the ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Profile-guided Optimization*** > ***Prepare application to create new profile*** setting.

   b. The `-prof-hw` switch or choose the ***Gather profile using hardware*** setting.

   This creates an executable file containing the necessary instrumentation for gathering profile data when run on hardware.

   For best results, click ***General*** (under ***Compiler***) in the tree control and select ***Enable optimization***/`-O` switch (`-O[0|1]`) or ***Interprocedural optimization***/`-ipa` (`-ipa`) switch.

2. Gather the profile. Run the executable on the hardware, with one or more training data sets. These training data sets should be representative of the data that you expect the application to process in the field. Note that unrepresentative training data sets can cause performance degradations when the application is used on real data. The profile is stored in files with the extension `.pgo` and `.pgt`.

3. Recompile the application using this gathered profile data:

   a. Turn off the `-prof-hw` switch or choose the ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Profile-guided Optimization*** > ***Gather profile using hardware*** setting.

   b. Turn off the `-pguide` switch or the ***Prepare application to create new profile*** setting.

   c. Place the `.pgo` file on the command line or include it in the list of profiles under Optimize using existing profiles. The `.pgo` file contains a reference to the `.pgt` file, so this automatically includes the `.pgt` file.

   d. Ensure optimization is enabled:

   Click ***General*** (under ***Compiler***) in the tree control and select ***Enable optimization***/`-O` switch (`-O[0|1]`) and/or ***Interprocedural optimization***/`-ipa` (`-ipa`) switch.

PGO for hardware works by planting function calls into your application which record the execution count (and in multi-threaded cases, the thread identifier) at certain points. Applications built with PGO for hardware should be used for development and should not be released.

PGO for hardware requires that an I/O device is available in the application to produce its profiling data. The default I/O device will be used to perform I/O operations for PGO.

PGO for hardware flushes any remaining profile data still pending when `exit()` is invoked. Multi-threaded applications may need to flush data explicitly.

When C/C++ source files are specified in a compiler command line, any specified `.pgo` files will be used to guide compilation. However, any recompilation due to `.doj` files provided on the command line will reread the same `.pgo` file as when the source was previously compiled.

For example, `prof2.pgo` is ignored in the following commands:

```
cc21k -O f2.c -o f2.doj prof1.pgo
cc21k -o prog.dxe f1.asm f2.doj prof2.pgo
```

Flushing PGO Data in Multi-Threaded and Non-Terminating Applications

Applications profiled with profile-guided optimization for hardware need to ensure that the profiling information is flushed to the host machine. Flushing occurs when any of the following conditions are met:

- In an application linked with the single-threaded runtime libraries, data is flushed when the profile-guided optimization data buffer is full.
- In an application linked with the threadsafe runtime libraries, once the profile-guided optimization data buffer is 75% full, data will be flushed at the next available opportunity.
- When the profile-guided optimization maximum flush interval has been exceeded. By default the maximum flush interval is 10 minutes.
- When the application explicitly requests a flush of the profile-guided optimization data.

Multi-threaded applications and applications that do not terminate must be modified to flush the data at an appropriate time. To request a flush of the data, add a call to the function `pgo_hw_request_flush()`. The example code in the ***Flushing Profile-Guided Optimization Data From an Application*** example shows a function that has been modified to flush the profile-guided data. The required changes are conditionally included when the preprocessor macro `_PGO_HW` is defined. The `_PGO_HW` macro is only defined when the application is compiled with the `-pguide` and `-prof-hw` compiler switches. Flushing the data to the host is a cycle-intensive operation, so you should consider carefully where to place the call to flush within your application.

In the example, the flush request has been placed in function `do_pgo_flush()`, which is called after the critical data loop in an attempt to reduce the impact of the profiling on the application's behavior. `do_pgo_flush()` is marked by `#pragma pgo_ignore`, so that no profile information is generated for the function. Isolating the flushing action in this manner is important because the verifies that a gathered profile matches the function's structure, before using the profile in optimization; if `pgo_hw_request_flush()` was conditionally called directly from `main_loop()`, when the application was recompiled with the gathered profile, but without the `-prof-hw` switch, the compiler would see that the call was now absent, making the profile invalid, and causing the optimizer to disregard the profile.

Example. Flushing Profile-Guided Optimization Data From an Application

```
#if defined(_PGO_HW)
   #include <pgo_hw.h>
#endif

extern int get_task(void);

#pragma pgo_ignore
static void do_pgo_flush(void) {
#if defined(_PGO_HW)
   pgo_hw_request_flush();
#endif
}
void main_loop(void) {
   while ( 1 ) {
      int task = get_task();
      if ( task == 1 ) {
         // perform critical data loop
         do_pgo_flush();
      } else {
         // other tasks
      }
   }
}
```

Profile-Guided Optimization and Multiple Source Uses

In some applications, it is convenient to build the same source file in more than one way within the same application. For example, a source file might be conditionally compiled with different macro settings. Alternatively, the same file might be compiled once, but linked more than once into the same application in a multi-core or multi-processor environment. In such circumstances, the typical behaviors of each instance in the application might differ. Identify and build the instances separately so that they can be profiled individually and optimized according to their typical use.

The `-pgo-session` *`session-id`* switch or ***PGO session name*** option (***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Profile-guided Optimization***) is used to separate profiles in such cases. It is used during both stage 1, where the compiler instruments the generated code for profiling, and during stage 3, where the compiler uses gathered profiles to guide the optimization.

During stage 1, when the compiler instruments the generated code, if the `-pgo-session` switch is used, then the compiler marks the instrumentation as belonging to the session *`session-id`*.

During stage 3, when the compiler reads gathered profiles, if the `-pgo-session` switch is used, then the compiler ignores all profile data not generated from code that was marked with the same *`session-id`*.

Therefore, the compiler can optimize each variant of the source's build according to how the variant is used, rather than according to an average of all uses.

Profile-Guided Optimization and the `-Ov num` Switch

When a `.pgo` file is placed on the command line, the optimization (`-O`) switch, by default, tries to balance between code performance and code-size considerations. It is equivalent to using the `-Ov 50` switch. To optimize solely for performance while using PGO, use the `-Ov 100` switch. The `-Ov num` switch is discussed further along with optimization for space in Smaller Applications: Optimizing for Code Size.

Profile-Guided Optimization and Multiple PGO Data Sets

When using profile-guided optimization with an executable constructed from multiple source files, the use of multiple PGO data sets will result in the creation of a temporary PGO information file (`.pgi`). This file is used by the compiler and prelinker to ensure that temporary PGO files can be recreated and to identify cases where objects and PGO data sets are invalid.

The compiler reports an error if any of the PGO data files have been modified between the initial compilation of an object and any recompilation that occurs at the final link stage. To avoid this error, perform a full recompilation after running the application to generate `.pgo` data files.

When to Use Profile-Guided Optimization

PGO should be performed as the last optimization step. If the application source code is changed after gathering profile data, this profile data becomes invalid. The compiler does not use profile data when it can detect that it is inaccurate. However, it is possible to change source code in a way that is not detectable to the compiler (for example, by changing constants). You should ensure that the profile data used for optimization remains accurate.

For more details on PGO, refer to Optimization Control.

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 optimizing. For example, it helps to know what data can be referenced by pointer parameters or if a variable actually has a constant value. The `-ipa` compiler switch enables interprocedural analysis (IPA), which can make this information available. When this switch is used, the compiler is called again from the link phase to recompile the program using additional information obtained during previous compilations.

This gathered information is stored within the object file generated during initial compilation. IPA retrieves the gathered information from the object file during linking, and uses it to recompile available source files where beneficial. Because recompilation is necessary, IPA-built modules in libraries can contribute to the optimization of application sources, but do not themselves benefit from IPA, as their source is not available for recompilation.

Because it only operates at link time, the effects of IPA are not seen if you compile with the `-S` switch. To see the assembly file when IPA is enabled, use the `-save-temps` switch, 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 `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 the aligned() Builtin and Using Pragmas for Optimization.

The volatile Type Qualifier

The `volatile` type qualifier is used to inform the compiler that it may not make any assumptions about a variable or memory location (or a series of them), and that such variables must be read from or written to as specified and in the same order as in the source code.

Failure to use `volatile` when necessary is a common programming error that can cause an application to fail when built in Release configuration with compiler optimizations enabled. This is because the compiler assumes that all non-volatile memory is modified explicitly and does not change in a way the compiler cannot see. This assumption is used extensively during optimization, where values held in memory may not be reloaded if they do not appear to have changed. Since the cases listed below do not adhere to the compiler's assumptions, the compiler must be informed of these situations through the use of the `volatile` type qualifier.

It is essential to make the following types of objects `volatile`-qualified in your application source:

- An object that is a memory-mapped register (MMR) or a memory-mapped device.
- An object that is shared between multiple concurrent threads of execution. This includes data that is shared between processors or data written by DMA.
- An object that is modified by an asynchronous event handler (for example, a global variable modified by an interrupt handler).
- An automatic storage duration object (i.e. a local variable declared on the stack) declared in a function that calls `setjmp()` and whose value is changed between the call to `setjmp()` and a corresponding call to `longjmp()`.

Data Types

The ***Scalar Data Types*** table shows the following scalar data types that the compiler supports.

Table 3-2: Scalar Data Types

Data Type	*Value*
Single-Word Fixed-Point Data Types: Native Arithmetic	
`char`	8-bit signed integer if compiling with `-char-size-8` on processors that support byte-addressing, else 32-bit signed integer
`unsigned char`	8-bit unsigned integer if compiling with `-char-size-8` on processors that support byte-addressing, else 32-bit unsigned integer
`short`	16-bit signed integer if compiling with `-char-size-8` on processors that support byte-addressing, else 32-bit signed integer
`unsigned short`	16-bit unsigned integer if compiling with `-char-size-8` on processors that support byte-addressing, else 32-bit unsigned integer
`int`	32-bit signed integer
`unsigned int`	32-bit unsigned integer

Table 3-2: Scalar Data Types (Continued)

Data Type	*Value*
`long`	32-bit signed integer
`unsigned long`	32-bit unsigned integer
Double-Word Fixed-Point Data Types: Emulated Arithmetic	
`long long`	64-bit signed integer
`unsigned long long`	64-bit unsigned integer
Single-Word Fixed-Point Data Types: Native Arithmetic	
`fract`	32-bit signed fractional
`unsigned fract`	32-bit unsigned fractional
`short fract`	32-bit signed fractional
`unsigned short fract`	32-bit unsigned fractional
`long fract`	32-bit signed fractional
`unsigned long fract`	32-bit unsigned fractional
Floating-Point Data Types: Native Arithmetic	
`float`	32-bit floating point
`double`	32-bit floating point *Note:* Applies when the ***Double size*** option is set to 32 bits, or the `-double-size-32` switch is used.
Floating-Point Data Types: Native Arithmetic only on processors that support native double-precision floating-point arithmetic (see the ***Processors Support Native Double-Precision Floating-Point Arithmetic*** table in Preprocessor Features). On all other processors, operations use Emulated Arithmetic.	
`double`	64-bit floating-point *Note:* Applies when the ***Double size*** option is set to 64 bits, or the `-double-size-64` switch is used.
`long double`	64-bit floating-point

Avoiding Emulated Arithmetic

Arithmetic operations for some data types are implemented by library functions because the processor hardware does not directly support these types. Consequently, operations for these 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.

The hardware does not provide direct support for division, so division and modulus operations are almost always multi-cycle operations, even on integral type inputs. If the compiler has to issue a full-division operation, it usually

needs to call a library function. One instance in which a library call is avoided is for integer division when the divisor is a compile-time constant and is a power of two. In 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 the 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.

Avoid emulated arithmetic operators where possible, especially in loops, where their use can inhibit more advanced optimization techniques, such as vectorization.

Getting the Most From IPA

Interprocedural analysis (IPA) is designed to try to propagate information about the program to parts of the optimizer that can use it. This section looks at what information is useful, and how to structure your code to make this information easily accessible for analysis.

The performance features are:

- Initialize Constants Statically
- Dual Word-Aligning Your Data
- Using the aligned() Builtin
- Avoiding Aliases

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,

```
// BAD: IPA cannot see that val is a constant
#include <stdio.h>
int val;  // initialized to zero

void init() {
   val = 3;  // re-assigned
}

void func() {
   printf("val %d",val);
}

int main() {
```

```
   init();
   func();
}
```

The code is better written as

```
// GOOD: IPA knows val is 3.
#include <stdio.h>
const int val = 3; // initialized once

void init() {
}

void func() {
   printf("val %d",val);
}
int main() {
   init();
   func();
}
```

Dual Word-Aligning Your Data

To make most efficient use of the hardware, it must be kept fed with data. In many algorithms, the balance of data accesses to computations is such that, to keep the hardware fully utilized, data must be fetched with loads wider than 32 bits.

For external data, the ADSP-2116x chips require that dual-word memory accesses reference dual-word-aligned addresses. Therefore, for the most efficient code generation, ensure that your 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. However, arrays within structures are not aligned beyond the required alignment for their type. It may be worth using the `#pragma align` directive to force the alignment of arrays in this case.

If you write programs that pass only the address of the first element of a global array as a parameter, and loop that process through these input arrays, an element at a time (starting at element zero), then IPA should be able to establish that the alignment is suitable for full-width accesses.

Where an inner loop processes a single row of a multi-dimensional array, try to ensure that each row begins on a two-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 the aligned() Builtin

To avoid the need to use IPA to propagate alignment, and for situations when IPA cannot guarantee the alignment (but you can), use the `aligned()` built-in function to assert the alignment of important pointers, meaning that the pointer points to data that is aligned. Remember when adding this declaration that you are responsible for

making sure it is valid, and that if the assertion is not true, the code produced by the compiler is likely to malfunction.

The assertion is particularly useful for function parameters, although you may assert that any pointer is aligned. For example, when compiling the function:

```
// BAD: without IPA, compiler doesn't know the alignment of a and b.
void copy(int *a, int *b) {
   int i;
   for (i=0; i<100; i++)
      a[i] = b[i];
}
```

the compiler does not know the alignment of pointers `a` and `b` if IPA is not being used. However, by modifying the function to:

```
// GOOD: both pointer parameters are known to be aligned.
#include <builtins.h>
void copy(int *a, int *b) {
   int i;
   aligned(a, 2 * sizeof(int));
   aligned(b, 2 * sizeof(int));
   for (i=0; i<100; i++)
       a[i] = b[i];
}
```

the compiler can be told that the pointers are aligned on dual-word boundaries. To assert instead that both pointers are always aligned one `int` before a dual-word boundary, use:

```
// GOOD: both pointer parameters are known to be misaligned.
#include <builtins.h>
void copy(int *a, int *b) {
   int i;
   aligned(a+1, 2 * sizeof(int));
   aligned(b+1, 2 * sizeof(int));
   for (i=0; i<100; i++)
       a[i] = b[i];
}
```

The expression used as the first parameter to the built-in function obeys the usual C rules for pointer arithmetic. The second parameter should give the alignment in addressable units (i.e. bytes when compiling with `-char-size-8` on processors that support byte-addressing, else in words) as a literal constant.

Avoiding Aliases

It may seem that the iterations can be performed in any order in the following loop:

```
// BAD: a and b may alias each other.
void fn(char a[], char b[], int n) {
   int i;
   for (i = 0; i < n; ++i)
```

```
        a[i] = b[i];
}
```

but a and b are both parameters, and, although they are declared with [], they are pointers that may point to the same array. When the same data may be reachable through two pointers, they are said to alias each other.

If IPA is enabled, the compiler looks at the call sites of fn and tries to determine whether a and b can ever point to the same array.

Even with IPA, it is easy to create what appears to the compiler as an alias. The analysis works by associating pointers with sets of variables that they may refer to at some point in the program. If the sets for two pointers intersect, then both pointers are assumed to point to the union of the two sets.

If fn above were called only in two places, with global arrays as arguments, then IPA would have the results shown below:

```
// GOOD: sets for a and b do not intersect: a and b are not aliases.
   fn(glob1, glob2, N);
   fn(glob1, glob2, N);
// GOOD: sets for a and b do not intersect: a and b are not aliases.
   fn(glob1, glob2, N);
   fn(glob3, glob4, N);
// BAD: sets intersect - both a and b may access glob1;  // a and b may be
aliases.
   fn(glob1, glob2, N);
   fn(glob3, glob1, N);
```

The third case arises because IPA considers the union of all calls at once, rather than considering each call individually, when determining whether there is a risk of aliasing. If each call were considered individually, IPA would have to take flow control into account and the number of permutations would significantly lengthen compilation time.

The lack of control flow analysis can also create problems when a single pointer is used in multiple contexts. For example, it is better to write

```
// GOOD: p and q do not alias.
int *p = a;
int *q = b;
   // some use of p
   // some use of q
```

than

```
// BAD: uses of p in different contexts may alias.
int *p = a;
   // some use of p
p = b;
   // some use of p
```

because the latter may cause extra apparent aliases between the two uses.

Indexed Arrays Versus Pointers

The C language allows a program to access data from an array in two ways: either by indexing from an invariant base pointer, or by incrementing a pointer. The following two versions of vector addition illustrate the two styles:

Style 1:Using indexed arrays(indexing from a base pointer)

```
void va_ind(const short a[], const short b[], short out[], int n)
{
   int i;
   for (i = 0; i < n; ++i)
      out[i] = a[i] + b[i];
}
```

Style 2:Incrementing a pointer

```
void va_ptr(const short a[], const short b[], short out[], int n)
{
   int i;
   short *pout = out;
   const short *pa = a, *pb = b;
   for (i = 0; i < n; ++i)
      *pout++ = *pa++ + *pb++;
}
```

Trying Pointer and Indexed Styles

One might hope that the chosen style would not 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.

NOTE: Try both pointer and index styles.

The pointer style introduces additional variables that compete with the surrounding code for resources during the compiler optimizer's analysis. Array accesses, on the other hand, must be transformed to pointers by the compiler, and sometimes this is accomplished better by hand.

The best strategy is to start with array notation. If the generated code looks unsatisfactory, try using pointers. Outside the critical loops, use the indexed style, since it is easier to understand.

Using Function Inlining

Function inlining may be used in two ways:

- By annotating functions in the source code with the `inline` keyword. In this case, function inlining is performed only when optimization is enabled.
- By turning on automatic inlining with the `-Oa` switch or the ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Inlining*** option to ***Automatic, automatically enabling optimization***.

NOTE: 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.

```
// GOOD: use of the inline keyword.
inline int add(int a, int b) {
   return (a+b);
}
```

Inlining has a code-size-to-performance trade-off that should be considered. With `-Oa`, the compiler automatically inlines small functions where possible. If the application has a tight upper code-size limit, the resulting code-size expansion may be too great. Consider using automatic inlining in conjunction with the `-Ov` *`num`* switch or the ***Optimize for code speed/size*** slider to restrict inlining (and other optimizations with a code-size cost) to parts of the application that are performance-critical. It is discussed in more detail later in this chapter.

For more information, see Function Inlining.

Using Inline asm Statements

The compiler allows use of inline `asm` statements to insert small sections of assembly into C code.

NOTE: Avoid the 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 a large number of built-in functions that generate specific hardware instructions. These are designed to allow the programmer to more finely tune the code produced by the compiler, or to allow access to system support functions. A complete list of compiler's built-in functions is given in Compiler Built-In Functions.

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 by the compiler.

Examples of efficient use of built-in functions are given in Compiler Built-In Functions.

For more information, see Inline Assembly Language Support Keyword (asm).

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.

NOTE: 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, the processor stalls.

Consider the dot product loop below. Because data is loaded from both array a and array b in every iteration of the loop, it may be useful to ensure that these arrays are located in different blocks.

```
// BAD: compiler assumes that two memory accesses together may give a stall.
for (i=0; i<100; i++)
   sum += a[i] * b[i];
```

The "Dual Memory Support Language Keywords" compiler extension (see Dual Memory Support Keywords (pm dm)) can improve the compiler's use of the memory system. Placing a pm qualifier before the type definition tells the compiler that the array is located in what is referenced as "Program Memory" (pm).

The memory of the SHARC processor is in one unified address space and there is no restriction on where in 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".

To allow simultaneous accesses to the two buffers, modify the array declaration of either a or b program by adding the pm qualifier. Also add the pm qualifier to the declarations of any pointers that point to the pm buffer.

For example,

```
pm int a[100];
```

and any pointers to the buffer a become, for example,

```
pm int *p = a;
```

Note that only global or static data can be explicitly placed in Program Memory.

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 optimization of conditional branches. 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 `buffer_valid()` would usually return true, but that `send_msg()` would rarely do so, the code could be written as

```
if (expected_true(buffer_valid(data_buffer)))
   if (expected_false(send_msg(data_buffer)))
      system_failure();
```

Example of Compiler Performance Built-in Functions

The following example project demonstrates the use of these compiler performance built-in functions:

`SHARC\Examples\No_HW_Required\ADSP-21469\Branch_Prediction`

The project loops through a section of character data, counting the different types of characters it finds. It produces three overall counts: lowercase letters, uppercase letters, and non-alphabetic characters. The effective test is as follows:

```
if (isupper(c))
   nAZ++;   // count one more uppercase letter
else if (islower(c))
   naz++;   // count one more lowercase letter
else
   nx++;    // count one more non-alphabetic character
```

The performance of the application is determined by the compiler's ability to correctly predict which of these two tests is going to evaluate as true most frequently.

In the source code for this example, the two tests are enclosed in two macros, `EXPRA(c)` and `EXPRB(c)`:

```
if (EXPRA(isupper(c)))
   nAZ++;   // count one more uppercase letter
else if (EXPRB(islower(c)))
   naz++;   // count one more lowercase letter
else
   nx++;    // count one more non-alphabetic character
```

The macros are defined conditionally according to the macro `EXPRS`, at compile-time, as shown by the ***How Macro EXPRS Affects Macros EXPRA and EXPRB*** table. By setting `EXPRS` to different values, you can see the effect the compiler performance built-in functions have on the application's overall performance. By leaving the `EXPRS` macro undefined, you can see how the compiler's default heuristics compare.

Table 3-3: How Macro EXPRS Affects Macros EXPRA and EXPRB

Value of EXPRS	*EXPRA Expected To Be*	*EXPRB Expected To Be*
Undefined	No prediction	No prediction
1	True	True

Table 3-3: How Macro EXPRS Affects Macros EXPRA and EXPRB (Continued)

Value of EXPRS	*EXPRA Expected To Be*	*EXPRB Expected To Be*
2	False	True
3	True	False
4	False	False

To use the example, do the following:

1. Import the `Branch_Prediction` project into your workspace:
   a. Select ***File*** > ***Import***.
   b. Choose ***General*** > ***Existing Projects Into Workspace*** and click ***Next***.
   c. Ensure ***Select root directory*** is checked.
   d. Browse to the `SHARC\Examples\No_HW_Required\ADSP-21469\Branch_Prediction` directory.

      Click ***OK***.
   e. Check the Branch_Prediction project.
   f. Ensure ***Copy projects into workspace*** is checked.
   g. Click ***Finish***.
2. Build the project.
3. Create a launch configuration for the ADSP-21469 SHARC processor, for the executable you have just built.
4. Launch the configuration, and run the executable to completion.

   You will see some output on the console as the project reports the number of characters of each type found in the string. The application will also report the number of cycles used.
5. Open ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Preprocessor***.
6. In the ***Preprocessor definitions*** field, add `EXPRS=1`.

   Click ***OK***.
7. Re-build and re-run the application.

   You will receive the same counts from the application, but the cycle counts will be different.
8. Try using values 2, 3, or 4 for `EXPRS` instead, and determine which combination of `expected_true()` and `expected_false()` built-in functions produces the best performance.

See Compiler Performance Built-In Functions (on `expected_true` and `expected_false` functions) for more information.

Using PGO in Function Profiling

The compiler can also determine the most commonly-executed branches automatically, using profile-guided optimization (PGO). See Optimization Control for more details.

Example of Using Profile-Guided Optimization

Continuing with the same example (in Improving Conditional Code), PGO can determine the best settings for the branches in `EXPRA(c)` and `EXPRB(c)` (and all other parts of the source code) using profiling.

NOTE: Normally, when using PGO, you would configure one or more input files as part of your data set. The application would read its inputs from these files, via the peripherals the application uses, and the data would influence the gathered profile. For this example, all the input data is embedded in the application source.

Opening the Project

To use the example, do the following:

1. Import the `Branch_Prediction` project into your workspace:
   a. Select ***File*** > ***Import***.
   b. Choose ***General*** > ***Existing Projects Into Workspace*** and click ***Next***.
   c. Ensure ***Select root directory*** is checked.
   d. Browse to the `SHARC\Examples\No_HW_Required\ADSP-21469\Branch_Prediction` directory.

      Click ***OK***.
   e. Check the `Branch_Prediction` project.
   f. Ensure ***Copy projects into workspace*** is checked.
   g. Click ***Finish***.
2. Ensure that the ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Preprocessor*** > ***Preprocessor definitions*** field does *not* contain a definition for `EXPRS`.
3. Build the project.
4. Create a launch configuration for the ADSP-21469 SHARC processor, for the executable you have just built.
5. Launch the configuration, and run the executable to completion. You will see some output on the console as the project reports the number of characters of each type found in the string. The application will also report the number of cycles used.

Gathering the Profile

To gather the profile on a simulator launch configuration:

1. ***Select Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Profile-guided Optimization*** > ***Prepare application to create new profile.***
2. In your launch configuration, go to the ***Automatic Breakpoints*** tab, and add a new software breakpoint on the label `___lib_start`.
3. Build the application, and launch it.
4. When the start breakpoint is reached, select ***Target*** > ***PGO*** > ***Simulator*** > ***Start***.
5. Continue running the application, until it reaches the `__lib_prog_term label`.
6. Select ***Target*** > ***PGO*** > ***Simulator*** > ***Stop and Save***.

Because the application is running on a simulator, the simulator does the work of gathering the profile, so the cycle-count will be the same as before.

To gather the profile on a hardware launch configuration:

1. Select ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** >***Tool Settings*** > ***Compiler*** > ***Profile-guided Optimization*** > ***Gather profile using hardware*** and ***Prepare application to create new profile***.
2. Build the application, then launch it.
3. Continue running the application, until it reaches the `__lib_prog_term` label.

Because the application is running on hardware, the compiler has planted additional code to gather the profile, so the cycle-count reported will be considerably higher than before. This is not a concern.

Rebuilding With the Profile

The profile will have been gathered into the file `Debug/Branch_Prediction.pgo`, within your project's directory. You now need to rebuild the application using this profile, telling the compiler to optimize the application according to execution counts for each path in the program. To do this:

1. Ensure ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Profile-guided Optimization*** > ***Gather profile using hardware*** is *not* selected.
2. Ensure ***Prepare application to create new profile*** is not selected.
3. Select ***Optimize using existing profiles***.
4. Add `Debug/Branch_Prediction.pgo` to the list of ***Profiles***.
5. Click ***General*** (under ***Compiler***) in the tree control and ensure ***Enable Optimization*** is selected.
6. Rebuild the application.

Now relaunch and run your rebuilt application. You will see a lower cycle count than first reported, as the compiler has rearranged the generated code so that the most commonly-executed paths are the defaults.

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.

This section describes:

- Keeping Loops Short
- Avoiding Unrolling Loops
- Avoiding Loop-Carried Dependencies
- Avoiding Loop Rotation by Hand
- Avoiding Complex Array Indexing
- Inner Loops Versus Outer Loops
- Avoiding Conditional Code in Loops
- Avoiding Placing Function Calls in Loops
- Avoiding Non-Unit Strides
- Loop Control
- Using the Restrict Qualifier

Keeping Loops Short

For best code efficiency, loops should be short. Large loop bodies are usually more complex and difficult to optimize. Large loops may also require register data to be stored in memory, which decreases code density and execution performance.

Avoiding Unrolling Loops

Do not unroll loops yourself. Not only does loop unrolling make the program harder to read but it also prevents optimization by complicating the code for the compiler.

```
// GOOD: the compiler unrolls if it helps.
void va1(const int a[], const int b[], int c[], int n) {
   int i;
   for (i = 0; i < n; ++i) {
      c[i] = b[i] + a[i];
   }
}
// BAD: harder for the compiler to optimize.
void va2(const int a[], const int b[], int c[], int n) {
   int xa, xb, xc, ya, yb, yc;
   int i;
   for (i = 0; i < n; i+=2) {
      xb = b[i]; yb = b[i+1];
```

```
      xa = a[i]; ya = a[i+1];
      xc = xa + xb; yc = ya + yb;
      c[i] = xc; c[i+1] = yc;
   }
}
```

Avoiding Loop Rotation by Hand

Do not rotate loops by hand. Programmers are often tempted to "rotate" loops in DSP code by hand, attempting to execute loads and stores from earlier or future iterations at the same time as computation from the current iteration. This technique introduces loop-carried dependencies that prevent the compiler from rearranging the code effectively. It is better to give the compiler a simpler version, and leave the rotation to the compiler.

For example,

```
// GOOD: is rotated by the compiler.
int ss(int *a, int *b, int n) {
   int sum = 0;
   int i;
   for (i = 0; i < n; i++) {
      sum += a[i] + b[i];
   }
   return sum;
}
// BAD: rotated by hand: hard for the compiler to optimize.
int ss(int *a, int *b, int n) {
   int ta, tb;
   int sum = 0;
   int i = 0;
   ta = a[i]; tb = b[i];
   for (i = 1; i < n; i++) {
      sum += ta + tb;
      ta = a[i]; tb = b[i];
   }
   sum += ta + tb;
   return sum;
}
```

Rotating the loop required adding the scalar variables `ta` and `tb` and introducing loop-carried dependencies.

Avoiding Complex Array Indexing

Other dependencies can be caused by writes to array elements. In the following loop, the optimizer cannot determine whether the load from `a` reads a value defined on a previous iteration or one that is overwritten in a subsequent iteration.

```
// BAD: has array dependency.
for (i = 0; i < n; ++i)
   a[i] = b[i] * a[c[i]];
```

The optimizer can resolve access patterns where the addresses are expressions that vary by a fixed amount on each iteration. These are known as *induction variables*.

```
// GOOD: uses induction variables.
for (i = 0; i < n; ++i)
   a[i+4] = b[i] * a[i];
```

Inner Loops Versus Outer Loops

Inner loops should iterate more than outer loops.

The optimizer focuses on improving the performance of inner loops because this is where most programs spend the majority of their time. It is considered a good trade-off for an optimization to slow down the code before and after a loop to make the loop body run faster. Therefore, try to make sure that your algorithm also spends most of its time in the inner loop; otherwise it may actually run slower after optimization. If you have nested loops where the outer loop runs many times and the inner loop runs a small number of times, try to rewrite the loops so that it is the outer loop that has fewer iterations.

Avoiding Conditional Code in Loops

If a loop contains conditional code, control-flow latencies may incur large penalties if the compiler has to generate conditional jumps within the loop. In some cases, the compiler is able to convert `if-then-else` and `?:` constructs into conditional instructions. In other cases, it can evaluate the expression entirely outside of the loop. However, for important loops, linear code should be written where possible.

There are several techniques for removing conditional code. For example, there is hardware support for `min` and `max`. The compiler usually succeeds in transforming conditional code equivalent to `min` or `max` into the single instruction. With particularly convoluted code the transformation may be missed, in which case it is better to use `min` or `max` in the source code.

The compiler can sometimes perform the loop transformation that interchanges conditional code and loop structures. Nevertheless, instead of writing

```
// BAD: loop contains conditional code.
for (i=0; i<100; i++) {
   if (mult_by_b)
      sum1 += a[i] * b[i];
   else
      sum1 += a[i] * c[i];
}
```

it is better to write

```
// GOOD: two simple loops can be optimized well.
if (mult_by_b) {
   for (i=0; i<100; i++)
      sum1 += a[i] * b[i];
} else {
   for (i=0; i<100; i++)
```

```
      sum1 += a[i] * c[i];
}
```

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, operations such as division, modulus, and some type coercions may implicitly call library functions. These are expensive operations which you should try to avoid in inner loops. For more details, see Data Types.

Avoiding Non-Unit Strides

If you write a loop, such as

```
// BAD: non-unit stride means division may be required.
   for (i=0; i<n; i+=3) {
         // some code
   }
```

then for the compiler to turn this into a hardware loop, it needs to work out the loop trip count. To do so, it must divide `n` by 3. The compiler may decide that this is worthwhile as it speeds up the loop, but division is an expensive operation. Try to avoid creating loop control variables with strides other than 1 or -1.

In addition, try to keep memory accesses in consecutive iterations of an inner loop contiguous. This is particularly applicable to multi-dimensional arrays.

Therefore,

```
// GOOD: memory accesses contiguous in inner loop
for (i=0; i<100; i++)
   for (j=0; j<100; j++)
      sum += a[i][j];
```

is likely to be better than

```
// BAD: loop cannot be unrolled to use wide loads.
for (i=0; i<100; i++)
   for (j=0; j<100; j++)
      sum += a[j][i];
```

as the former is more amenable to vectorization.

Loop Control

Use `int` types for loop control variables and array indices. For loop control variables and array indices, it is always better to use signed `ints` rather than any other integral type. For other integral types, the C standard requires various type promotions and standard conversions that complicate the code for the compiler optimizer. Frequently, the compiler is still able to deal with such code and create hardware loops and pointer induction variables. However, it 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. Use automatic variables for loop control and loop exit test. 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

```
// BAD: may need to reload globvar on every iteration.
for (i=0; i<globvar; i++)
   a[i] = a[i] + 1;
```

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 reloaded each time around the loop before performing the exit test.

In this circumstance, the programmer can make the compiler's job easier by writing:

```
// GOOD: easily becomes a hardware loop.
int upper_bound = globvar;
for (i=0; i<upper_bound; i++)
   a[i] = a[i] + 1;
```

Using the Restrict Qualifier

The `restrict` qualifier provides one way to help the compiler resolve pointer aliasing ambiguities. Accesses from distinct restricted pointers do not interfere with each other. The loads and stores in the following loop:

```
// BAD: possible alias of arrays a and b
void copy(int *a, int *b) {
   int i;
   for (i=0; i<100; i++)
      a[i] = b[i];
}
```

may be disambiguated by writing

```
// GOOD: restrict qualifier tells compiler that memory  // accesses do not alias
void copy(int * restrict a, int * restrict b) {
   int i;
   for (i=0; i<100; i++)
      a[i] = b[i];
}
```

The `restrict` keyword is particularly useful on function parameters. but it can be used on any variable declaration. For example, the `copy` function may also be written as:

```
void copy(int *a, int *b) {
int i;
int * restrict p = a;
int * restrict q = b;

for (i=0; i<100; i++)
   *p++ = *q++;
}
```

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, it presents some code examples where implementation choices are available to the programmer. For more information, refer to Compiler Built-In Functions.

Using System Support Built-In Functions

Built-in functions are provided 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.

Built-in functions cause the compiler to generate efficient inline instructions and their use often results in better optimization of the surrounding code at the point where they are used. Using built-in functions also usually results in improved code readability.

For more information on supported built-in functions, refer to Compiler Built-In Functions.

Examples of the two styles are:

```
// BAD: uses inline asm statement
asm("#include <def21160.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
}

// GOOD: uses sysreg.h
#include <sysreg.h>          // Sysreg functions
#include <def21160.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
}
```

This example calls a function with interrupts disabled.

Using Circular Buffers

Circular buffers are useful in DSP-style code. They can be used in several ways. Consider the C code:

```
// GOOD: the compiler knows that b is accessed as a circular buffer
for (i=0; i<1000; i++) {
   sum += a[i] * b[i%20];
}
```

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 this more complex example.

```
// BAD: may not be able to use circular buffer to access b
for (i=0; i<1000; i+=n) {
   sum += a[i] * b[i%20];
}
```

In this case, the compiler does not know if `n` is positive and less than 20. If it is, 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.

The first option is to compile with the `-force-circbuf` switch. 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 value of `i` must be positive.
- The value of `n` must be constant across the loop, and greater than zero (as the length of the buffer).
- The value of `a` must be a constant across the loop (as the base address of the circular buffer).
- The initial value of `i` must be such that `a[i]` refers a valid position within the circular buffer. This is because the circular buffer operations will take effect when advancing from position `a[i]` to either `a[i+m]` or `a[i-m]`, by addition or subtraction, respectively. If `a[i]` is not initially valid, then any access before the first advancement will not access the buffer, and `a[i+m]` and `a[i-m]` will not be guaranteed to reference the buffer after advancement.

NOTE: Circular buffer operations (which add or subtract the buffer length to a pointer) are semantically different from `a[i%n]` (which performs a modulo operation on an index, and then adds the result to a base pointer). If you use the `-force-circbuf` switch when the above conditions are not true, the compiler generates code that does not have the intended effect.

The second, and preferred, option is to use built-in functions to perform the circular buffering. Two functions (`circindex` and `circptr`) are provided for this purpose.

To make it clear to the compiler that a circular buffer should be used, you may write either:

```
// GOOD: explicit use of circular buffer via circindex
#include <builtins.h>
for (i=0, j=0; i<1000; i+=n) {
   sum += a[i] * b[j];
   j = circindex(j, n, 20);
}
```

or

```
// GOOD: explicit use of circular buffer via circptr
#include <builtins.h>
int *p = b;
for (i=0; i<1000; i+=n) {
   sum += a[i] * (*p);
   p = circptr(p, n * sizeof(*p), b, 20 * sizeof(*p));
}
```

For more information, refer to Compiler Built-In Functions.

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 clear 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`, 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) can be helpful; see `-flags-{asm|compiler|ipa|lib|link|mem|prelink}` *`switch[, switch2[, ...]]`*. This operation performs section elimination in the linker to remove unneeded data and code. If the code produced with the `-Os` and `-flags-link -e` switches does not meet the code-size constraint, some analysis of the source code is required to try to 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*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** in the IDE). 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 optimize 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) 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.

NOTE: 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. 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 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 are normally 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,

```
// GOOD: uses #pragma alloc to disambiguate out from a and b
#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];
}
```

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 above example might further be refined if it is known that the `new_buf` function always returns buffers which are aligned on a dual-word boundary.

```
// GOOD: uses pragma result_alignment to specify that out has
// strict alignment
#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];
}
```

Since the alignment is specified in addressable units, when compiling with `-char-size-8` on processors that support byte-addressing, the alignment should be specified in bytes rather than words (i.e. `#pragma result_alignment (8)` in the above example). Further details on this pragma may be found in #pragma result_alignment (*n*). Another more laborious way to achieve the same effect would be to use `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 a simple assembly function, to add two integers and mask the result to fit into 8 bits (note that this code is intended to be called from code built using the `-char-size-32` switch, or for a processor that does not support byte-addressing):

```
_add_mask:
   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,

```
// GOOD: uses regs_clobbered to increase call-preserved register set.
   #pragma regs_clobbered "r0, r2, r8, i12, ASTAT"
   int add_mask(int, int);
```

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:

```
// BAD: function thought to clobber entire volatile register set
int add_mask(int a, int b) {
   return ((a+b)&255);
}
```

Since this function does not need many registers when compiled, it can be defined using:

```
// GOOD: function compiled to preserve most registers
#pragma regs_clobbered "r0, r2, i12, CCset"
```

```
int add_mask(int a, int b) {
   return ((a+b)&255);
}
```

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_call string.

#pragma optimize_{off|for_speed|for_space|as_cmd_line}

The `optimize_` pragmas may be used to change the optimization setting on a function-by-function basis. In particular, it may be useful to optimize functions that are rarely called (for example, error handling code) for space (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.

> **NOTE:** #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 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`, `SIMD_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. (A "guard" is code generated by the compiler to test a condition at runtime rather than at compilation time.) 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:

```
// GOOD: the loop_count pragma gives compiler helpful information
// to assist optimization)
#pragma loop_count(/*minimum*/ 40, /*maximum*/ 100, /*modulo*/ 4)
for (i=0; i<n; i++)
   a[i] = b[i];
```

For more information, refer to #pragma loop_count (*min, max, modulo*) .

#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, or directly before a function to disable vectorization for all loops in the function.

#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,

```
// BAD: cannot be vectorized due to possible alias between a and b
for (i=0; i<100; i++)
   a[i] = b[i] + a[i-4];
```

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.

```
// GOOD: pragma vector_for disambiguates alias
#pragma vector_for (4)
for (i=0; i<100; i++)
   a[i] = b[i] + a[i-4];
```

Note that this pragma does not force the compiler to vectorize the loop. The optimizer checks various properties of the loop and does not vectorize it if it believes that it is unsafe or 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.

#pragma SIMD_for

The `SIMD_for` pragma 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 #pragma SIMD_for.

#pragma all_aligned

The `all_aligned` pragma is used as shorthand for multiple `aligned()` assertions. By prefixing a `for` loop with the pragma, you can assert that every pointer variable in the loop is aligned on a dual-word boundary at the beginning of the first iteration.

Therefore, adding the pragma to the following loop

```
// GOOD: uses all_aligned to inform compiler of alignment of a and b
#pragma all_aligned
for (i=0; i<100; i++)
   a[i] = b[i];
```

is equivalent to writing

```
// GOOD: uses aligned() to give alignment of a and b
#include <builtins.h>
aligned(a, 2 * sizeof(*a));
aligned(b, 2 * sizeof(*b));
for (i=0; i<100; i++)
   a[i] = b[i];
```

In addition, the `all_aligned` pragma may take an optional literal integer argument `n` in parentheses. This tells the compiler that all pointer variables are aligned on a word boundary at the beginning of the nth iteration. Note that the iteration count begins at zero. Therefore,

```
// GOOD: uses all_aligned to inform compiler of alignment of a and b
#pragma all_aligned (1)
for (i=99; i>=0; i--)
    a[i] = b[i];
```

is equivalent to

```
// GOOD: uses aligned() to give alignment of a and b
#include <builtins.h>
aligned(a+98, 2 * sizeof(*a));
aligned(b+98, 2 * sizeof(*b));
for (i=99; i>=0; i--)
   a[i] = b[i];
```

For more information, refer to #pragma all_aligned and Using the aligned() Builtin.

#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 for more information.

Useful Optimization Switches

The ***C/C++ Compiler Optimization Switches*** table lists the compiler switches useful during the optimization process.

Table 3-4: C/C++ Compiler Optimization Switches

Switch Name	*Description*
-const-read-write	Specifies that data accessed via a pointer to const data may be modified elsewhere
-flags-link -e (see -flags-{asm|compiler|ipa|lib|link|mem|prelink} *switch[, switch2[, ...]]*)	Specifies linker section elimination
-force-circbuf	Treats array references of the form array[i%n] as circular buffer operations
-ipa	Turns on inter-procedural optimization. Implies use of -O. May be used in conjunction with -Os or -Ov.
-no-fp-associative	Does not treat floating-point multiply and addition as an associative
-no-saturation	Does not turn non-saturating operations into saturating ones
-O[0|1]	Enables code optimizations and optimizes the file for speed
-Os	Optimizes the file for size
-Ov *num*	Controls speed vs. size optimizations (sliding scale)
-pguide	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization
-save-temps	Saves intermediate files (for example, .s)

How Loop Optimization Works

Loop optimization is important to overall application performance, because any performance gain achieved within the body of a loop reaps a benefit for every iteration of that loop. This section provides an introduction to some of the concepts used in loop optimization, helping you to use the compiler features in this chapter.

This section contains:

- Terminology
- Loop Optimization Concepts
- A Worked Example

Terminology

This section describes terms that have particular meanings for compiler behavior.

Clobbered Register

A register is *clobbered* if its value is changed so that the compiler cannot usefully make assumptions about its new contents.

For example, when the compiler generates a call to an external function, the compiler considers all caller-preserved registers to be clobbered by the called function. Once the called function returns, the compiler cannot make any

assumptions about the values of those registers. This is why they are called "caller-preserved". If the caller needs the values in those registers, the caller must preserve them itself.

The set of registers clobbered by a function can be changed using `#pragma regs_clobbered`, and the set of registers changed by a gnu `asm` statement is determined by the clobber part of the `asm` statement.

Live Register

A register is *live* if it contains a value needed by the compiler, and thus cannot be overwritten by a new assignment to that register. For example, to do "A = B + C", the compiler might produce:

```
reg1 = load B        // reg1 becomes live
reg2 = load C        // reg2 becomes live
reg1 = reg1 + reg2   // reg2 ceases to be live;
                     // reg1 still live, but with a different
                     // value
store reg1 to A      // reg1 ceases to be live
```

Liveness determines which registers the compiler may use. In this example, since `reg1` is used to load `B`, and that register must maintain its value until the addition, `reg1` cannot also be used to load the value of `C`, unless the value in `reg1` is first stored elsewhere.

Spill

When a compiler needs to store a value in a register, and all usable registers are already live, the compiler must store the value of one of the registers to temporary storage (the stack). This *spilling* process prevents the loss of a necessary value.

Scheduling

Scheduling is the process of re-ordering the program instructions to increase the efficiency of the generated code but without changing the program's behavior. The compiler attempts to produce the most efficient schedule.

Loop Kernel

The *loop kernel* is the body of code that is executed once per iteration of the loop. It excludes any code required to set up the loop or to finalize it after completion.

Loop Prolog

A *loop prolog* is a sequence of code required to set the machine into a state whereby the loop kernel can execute. For example, the prolog may pre-load some values into registers ready for use in the loop kernel. Not all loops need a prolog.

Loop Epilog

A *loop epilog* is a sequence of code responsible for finalizing the execution of a loop. After each iteration of the loop kernel, the machine will be in a state where the next iteration can begin efficiently. The epilog moves values from the final iteration to where they need to be for the rest of the function to execute. For example, the epilog might save values to memory. Not all loops need an epilog.

Loop Invariant

A *loop invariant* is an expression that has the same value for all iterations of a loop. For example:

```
int i, n = 10;
for (i = 0; i < n; i++) {
    val += i;
}
```

The variable n is a loop invariant. Its value is not changed during the body of the loop, so n will have the value 10 for every iteration of the loop.

Hoisting

When the optimizer determines that some part of a loop is computing a value that is actually a loop invariant, it may move that computation to before the loop. This prevents the same value from being re-computed for every iteration. This is called *hoisting*.

Sinking

When the optimizer determines that some part of a loop is computing a value that is not used until the loop terminates, the compiler may move that computation to after the loop. This *sinking* process ensures the value is only computed using the values from the final iteration.

Loop Optimization Concepts

The compiler optimizer focuses considerable attention on program loops, as any gain in the loop's performance reaps the benefits on every iteration of the loop. The applied transformations can produce code that appears to be substantially different from the structure of the original source code. This section provides an introduction to the compiler's loop optimization, to help you understand why the code might be different.

This section describes:

- Software Pipelining
- Loop Rotation
- Loop Vectorization
- Modulo Scheduling

The following examples are presented in terms of a hypothetical machine. This machine is capable of issuing up to two instructions in parallel, provided one instruction is an arithmetic instruction, and the other is a load or a store. Two arithmetic instructions may not be issued at once, nor may two memory accesses:

```
t0 = t0 + t1;            // valid: single arithmetic
t2 = [p0];               // valid: single memory access
[p1] = t2;               // valid: single memory access
t2 = t1 + 4, t1 = [p0];  // valid: arithmetic and memory
t5 += 1, t6 -= 1;        // invalid: two arithmetic
[p3] = t2, t4 = [p5];    // invalid: two memory
```

The machine can use the old value of a register and assign a new value to it in the same cycle, for example:

```
t2 = t1 + 4, t1 = [p0];   // valid: arithmetic and memory
```

The value of `t1` on entry to the instruction is the value used in the addition. On completion of the instruction, `t1` contains the value loaded via the `p0` register.

The examples will show `START LOOP N` and `END LOOP`, to indicate the boundaries of a loop that iterates `N` times. (The mechanisms of the loop entry and exit are not relevant).

Software Pipelining

Software pipelining is analogous to hardware pipelining used in some processors. Whereas hardware pipelining allows a processor to start processing one instruction before the preceding instruction has completed, software pipelining allows the generated code to begin processing the next iteration of the original source-code loop before the preceding iteration is complete.

Software pipelining makes use of a processor's ability to multi-issue instructions. Regarding known delays between instructions, it schedules instructions from later iterations where there is spare capacity.

Loop Rotation

Loop rotation is a common technique of achieving software pipelining. It changes the logical start and end positions of the loop within the overall instruction sequence, to allow a better schedule within the loop itself. For example, this loop:

```
START LOOP N
   A
   B
   C
   D
   E
END LOOP
```

could be rotated to produce the following loop:

```
A
B
C
START LOOP N-1
   D
   E
   A
   B
   C
END LOOP
D
E
```

The order of instructions in the loop kernel is now different. It still circles from instruction `E` back to instruction `A`, but now it starts at `D`, rather than `A`. The loop also has a prolog and epilog added, to preserve the intended order of

instructions. Since the combined prolog and epilog make up a complete iteration of the loop, the kernel is now executing N-1 iterations, instead of N.

In this example, consider the following loop:

```
START LOOP N
   t0 += 1
   [p0++] = t0
END LOOP
```

This loop has a two-cycle kernel. While the machine could execute the two instructions in a single cycle-an arithmetic instruction and a memory access instruction-to do so would be invalid, because the second instruction depends upon the value computed in the first instruction.

However, if the loop is rotated, we get:

```
t0 += 1
START LOOP N-1
   [p0++] = t0
   t0 += 1
END LOOP
[p0++] = t0
```

The value being stored is computed in the previous iteration (or before the loop starts, in the prolog). This allows the two instructions to be executed in a single cycle:

```
t0 += 1
START LOOP N-1
   [p0++] = t0, t0 += 1
END LOOP
[p0++] = t0
```

Rotating the loop has presented an opportunity by which the kth iteration of the original loop is starting (t0 += 1) while the (k-1)th iteration is completing ([p0++] = t0), so rotation has achieved software pipelining, and the performance of the loop is doubled.

Notice that this process has changed the structure of the program slightly: suppose that the loop construct always executes the loop at least once; that is, it is a 1..N count. Then if N==1, changing the loop to be N-1 would be problematic. In this example, the compiler inserts a guard: a conditional jump around the loop construct for the circumstances where the compiler cannot guarantee that N > 1:

```
t0 += 1
IF N == 1 JUMP L1;
   START LOOP N-1
   [p0++] = t0, t0 += 1
END LOOP
L1:
   [p0++] = t0
```

Loop Vectorization

Loop vectorization is another transformation that allows the generated code to execute more than one iteration in parallel. However, vectorization is different from software pipelining. Where software pipelining uses a different ordering of instructions to get better performance, vectorization uses a different set of instructions. These vector instructions act on multiple data elements concurrently to replace multiple executions of each original instruction.

For example, consider this dot-product loop:

```
int i, sum = 0;
for (i = 0; i < n; i++) {
    sum += x[i] * y[i];
}
```

This loop walks two arrays, reading consecutive values from each, multiplying them and adding the result to the ongoing sum. This loop has these important characteristics:

- Successive iterations of the loop read from adjacent locations in the arrays.
- The dependency between successive iterations is the summation, a commutative and associative operation.
- Operations such as load, multiply and add are often available in parallel versions on embedded processors.

These characteristics allow the optimizer to vectorize the loop so that two elements are read from each array per load, two multiplies are done, and two totals maintained.

The vectorized loop would be:

```
t0 = t1 = 0
START LOOP N/2
t2 = [p0++] (Wide)     // load x[i] and x[i+1]
t3 = [p1++] (Wide)     // load y[i] and y[i+1]
t0 += t2 * t3 (Low), t1 += t2 * t3 (High)    // vector mulacc
END LOOP
t0 = t0 + t1           // combine totals for low and high
```

Vectorization is most efficient when all the operations in the loop can be expressed in terms of parallel operations. Loops with conditional control flow in them are rarely vectorizable, because the compiler cannot guarantee that the condition will evaluate in the same way for all the iterations being executed in parallel. However computations that use the `?:` operator may be vectorized in many circumstances.

Vectorization is also affected by data alignment constraints and data access patterns. Data alignment affects vectorization because processors often constrain loads and stores to be aligned on certain boundaries. While the unvectorized version will guarantee this, the vectorized version imposes a greater constraint that may not be guaranteed. Data access patterns affect vectorization because memory accesses must be contiguous. If a loop accessed every tenth element, for example, then the compiler would not be able to combine the two loads for successive iterations into a single access.

Vectorization divides the generated iteration count by the number of iterations being processed in parallel. If the trip count of the original loop is unknown, the compiler will have to conditionally execute some iterations of the loop.

NOTE: Vectorization and software pipelining are not mutually exclusive: the compiler may vectorize a loop and then use software pipelining to obtain better performance.

Modulo Scheduling

Loop rotation, as described earlier, is a simple software-pipelining method that can often improve loop performance, but more complex examples require a more advanced approach. The compiler uses a popular technique known as *modulo scheduling*, which can produce more efficient schedules for loops than simple loop rotation.

Modulo scheduling is used to schedule innermost loops without control flow. A modulo-scheduled loop is described using the following parameters:

- Initiation interval (II): the number of cycles between initiating two successive iterations of the original loop.
- Minimum initiation interval due to resources (res MII): a lower limit for the initiation interval (II); an II lower than this would mean at least one of the resources being used at greater capacity than the machine allows.
- Minimum initiation interval due to recurrences (rec MII): an instruction cannot be executed until earlier instructions on which it depends have also been executed. These earlier instructions may belong to a previous loop iteration. A cycle of such dependencies (a recurrence) imposes a minimum number of cycles for the loop.
- Stage count (SC): the number of initiation intervals until the first iteration of the loop has completed. This is also the number of iterations in progress at any time within the kernel.
- Modulo variable expansion unroll factor (MVE unroll): the number of times the loop has to be unrolled to generate the schedule without overlapping register lifetimes.
- Trip count: the number of times the loop kernel iterates.
- Trip modulo: a number that is known to divide the trip count.
- Trip maximum: an upper limit for the trip count.
- Trip minimum: a lower limit for the trip count.

Understanding these parameters will allow you to interpret the generated code more easily. The compiler's assembly annotations use these terms, so you can examine the source code and the generated instructions, to see how the scheduling relates to the original source. See Assembly Optimizer Annotations for more information.

Modulo scheduling performs software pipelining by:

- Ordering the original instructions in a sequence (for simplicity referred to as the *base schedule*) that can be repeated after an interval known as the *initiation interval* ("II");
- Issuing parts of the base schedule belonging to successive iterations of the original loop, in parallel.

For the purposes of this discussion, all instructions will be assumed to require only a single cycle to execute; on a real processor, stalls affect the initiation interval, so a loop that executes in II cycles may have fewer than II instructions.

Initiation Interval (II) and the Kernel

Consider the loop

```
START LOOP N
    A
    B
    C
    D
    E
    F
    G
    H
END LOOP
```

Now consider that the compiler finds a new order for `A,B,C,D,E,F,G,H` grouping; some of them on the same cycle so that a new instance of the sequence can be started every two cycles. Say this *base schedule* is given in the ***Base Schedule*** table, where `I1,I2,...,I8` are `A,B,...,H` reordered. Albeit a valid schedule for the original loop, the base schedule is not the final modulo schedule; it may not even be the shortest schedule of the original loop. However, the base schedule is used to obtain the modulo schedule, by being able to initiate it every II=2 cycles, as seen in the ***Obtaining the Modulo Schedule by Repeating the Base Schedule every II=2 Cycles*** table.

Table 3-5: Base Schedule

Cycle	*Instructions*
1	I1
2	I2, I3
3	I4, I5
4	I6
5	I7
6	I8

Table 3-6: Obtaining the Modulo Schedule by Repeating the Base Schedule Every II=2 Cycles

Cycle	*Iteration 1*	*Iteration 2*	*Iteration 3*	*Iteration 4*
1	I1			
2	I2, I3			
3	I4, I5	I1		
4	I6	I2, I3		
5	***I7***	***I4, I5***	***I1***	
6	***I8***	***I6***	***I2, I3***	
7		I7	I4, I5	I1
8		I8	I6	I2, I3
9			I7	I4, I5

Table 3-6: Obtaining the Modulo Schedule by Repeating the Base Schedule Every II=2 Cycles (Continued)

Cycle	*Iteration 1*	*Iteration 2*	*Iteration 3*	*Iteration 4*
10			I8	I6

Starting at cycle 5, the pattern in the ***Loop Kernel (N>=3)*** table keeps repeating every 2 cycles. This repeating pattern is the *kernel*, and it represents the modulo scheduled loop.

Table 3-7: Loop Kernel (N>=3)

Cycle	*Iteration N-2 (last stage)*	*Iteration N-1 (2nd stage)*	*Iteration N (1st stage)*
II*N-1	I7	I4, I5	I1
II*N	I8	I6	I2, I3

The initiation interval has the value II=2, because iteration `i+1` can start two cycles after the cycle on which iteration `i` starts. This way, one iteration of the original loop is initiated every II cycles, running in parallel with previous, unfinished iterations.

The initiation interval of the loop indicates several important characteristics of the schedule for the loop:

- The loop kernel will be II cycles in length.
- A new iteration of the original loop will start every II cycles. An iteration of the original loop will end every II cycles.
- The same instruction will execute on cycle `c` and on cycle `c+II` (hence the name modulo schedule).

Finding a modulo schedule implies finding a base schedule and an II such that the base schedule can be initiated every II cycles.

If the compiler can reduce the value for II, it can start the next iteration sooner, and thus increase the performance of the loop: The lower the II, the more efficient the schedule. However the II is limited by a number of factors, including:

- The machine resources required by the instructions in the loop.
- The data dependencies and stalls between instructions.

We'll examine each of these limiting factors.

Minimum Initiation Interval Due to Resources (Res MII)

The first factor that limits II is machine resource usage. Let's start with the simple observation that the kernel of a modulo scheduled loop contains the same set of instructions as the original loop.

Assume a machine that can execute up to four instructions in parallel. If the loop has 8 instructions, then it requires a minimum of 2 lines in the kernel, since there can be at most 4 instructions on a line. This implies II has to be at least 2, and we can tell this without having found a base schedule for the loop, or even knowing what the specific instructions are.

Consider another example where the original loop contains 3 memory accesses to be scheduled on a machine that supports at most 2 memory accesses per cycle. This implies at least 2 cycles in the kernel, regardless of the rest of the instructions.

Given a set of instructions in a loop, we can determine a lower bound for the II of any modulo schedule for that loop based on resources required. This lower bound is called the *Resource based Minimum Initiation Interval (Res MII)*.

Minimum Initiation Interval Due to Recurrences (Rec MII)

A less obvious limitation for finding a low II are cycles in the data dependencies between instructions.

Assume that the loop to be scheduled contains (among others) the instructions:

```
i3: t3=t1+t5;   // t5 carried from the previous iteration
i5: t5=t1+t3;
```

Assume each line of instructions takes 1 cycle. If `i3` is executed at cycle `c` then `t3` is available at cycle `c+1` and `t5` cannot be computed earlier than `c+1` (because it depends on `t3`), and similarly the next time we compute `t3` cannot be earlier than `c+2`. Thus if we execute `i3` at cycle `c`, the next time we can execute `i3` again cannot be earlier than `c+2`. But for any modulo schedule, if an instruction is executed at cycle `c`, the next iteration will execute the same instruction at cycle `c+II`. Therefore, II has to be at least 2 due to the circular data dependency path `t3->t5->t3`.

This lower bound for II, given by circular data dependencies (recurrences) is called the *Minimum Initiation Interval Due to Recurrences (Rec MII)*, and the data dependency path is called *loop carry path*. There can be any number of loop carry paths in a loop, including none, and they are not necessarily disjoint.

Stage Count (SC)

The kernel in the ***Loop Kernel (N>=3)*** table (see Initiation Interval (II) and the Kernel) is formed of instructions which belong to 3 distinct iterations of the original loop: {`I7,I8`} end the "oldest" iteration - in other words they belong to the iteration started the longest time before the current cycle; {`I4,I5,I6`} belong to the next oldest initiated iteration, and so on. {`I1,I2,I3`} are the beginning of the youngest iteration.

The number of iterations of the original loop in progress at any time within the kernel is called the *Stage Count* (SC). This is also the number of initiation intervals until the first iteration of the loop completes. In our example `SC=3`.

The final schedule requires peeling a few instructions (the prolog) from the beginning of the first iteration and a few instructions (the epilog) from the end of the last iteration in order to preserve the structure of the kernel. This reduces the trip count from `N` to `N-(SC-1)`:

```
I1;                                // prolog
I2,I3;                             // prolog
I4,I5,     I1;                     // prolog
I6,        I2,I3;                  // prolog
LOOP N-2                           // i.e. N-(SC-1), where SC=3
I7,        I4,I5,      I1;         // kernel
I8,        I6,         I2,I3;      // kernel
END LOOP
```

```
            I7,         I4, I5;  // epilog
            I8,         I6;      // epilog
                        I7;      // epilog
                        I8;      // epilog
```

Another way of viewing the modulo schedule is to group instructions into stages as in the ***Instructions Grouped into Stages*** table, where each stage is viewed as a vector of height `II=2` of instruction lists (that represent parts of instruction lines).

Table 3-8: Instructions Grouped into Stages

StageCount	*Instructions*
SC0	I1, I2, I3
SC1	I4, I5, I6
SC2	I7, I8

Now the schedule can be viewed as:

```
SC0                               // prolog
SC1    SC0                        // prolog
LOOP (N-2)                        // That is N-(SC-1), where SC=3
SC2    SC1    SC0                 // kernel
END LOOP
       SC2    SC1                 // epilog
              SC2                 // epilog
```

where, for example, SC2 SC1 is the 2 line vector obtained from concatenating the lists in `SC2` and `SC1`.

Variable Expansion and MVE Unroll

There is one more issue to address for modulo schedule correctness.

Consider the sequence of instructions in the ***Problematic Instance*** table.

Table 3-9: Problematic Instance

Generic Instruction	*Specific Instance*
I1	t1=[p1++]
I2	t2=[p2++]
I3	t3=t1+t5
I4	t4=t2+1
I5	t5=t1+t3

Table 3-9: Problematic Instance (Continued)

Generic Instruction	*Specific Instance*
I6	t6=t4*t5
I7	t7=t6*t3
I8	[p8++]=t7

The following table shows the base schedule that is an instance of the one in the ***Base Schedule*** table (see Initiation Interval (II) and the Kernel).

Table 3-10: Base Schedule (From the Base Schedule Table) Applied to the Instances in the Problematic Instance Table

1	t1=[p1++]
2	t2=[p2++],t3=t1+t5
3	t4=t2+1,t5=t1+t3
4	t6=t4*t5
5	t7=t6*t3
6	[p8++]=t7

The ***Modulo Schedule Broken by Overlapping Lifetimes of t3*** table shows the corresponding modulo schedule with `II=2`.

Table 3-11: Modulo Schedule Broken by Overlapping Lifetimes of t3

	Iteration 1	*Iteration 2*	*Iteration 3 ...*
1	t1=[p1++]		
2	t2=[p2++],t3=t1+t5		
3	t4=t2+1,t5=t1+t3	t1=[p1++]	
4	t6=t4*t5	t2=[p2++],t3=t1+t5	
5	***t7=t6*t3***	***t4=t2+1,t5=t1+t3***	***t1=[p1++]***
6	***[p8++]=t7***	***t6=t4*t5***	***t2=[p2++],t3=t1+t5***
7		t7=t6*t3	t4=t2+1,t5=t1+t3
8		[p8++]=t7	t6=t4*t5
9			t7=t6*t3
10			[p8++]=t7

However, there is a problem with the schedule in the ***Modulo Schedule Broken by Overlapping Lifetimes of t3*** table: `t3` defined in the fourth cycle (second column in the table) is used on the fifth cycle (first column); however, the intended use was of the value defined on the second cycle (first column). In general, the value of `t3` used by `t7=t6*t3` in the kernel will be the one defined in the previous cycle, instead of the one defined 3 cycles earlier, as

intended. Thus, if the compiler were to use this schedule as-is, it would be clobbering the live value in `t3`. The lifetime of each value loaded into `t3` is 3 cycles, but the loop's initiation interval is only 2, so the lifetimes of `t3` from different iterations overlap.

The compiler fixes this by duplicating the kernel as many times as needed to exceed the longest lifetime in the base schedule, then renaming the variables that clash-in this case, just `t3`. In the ***Modulo Schedule Corrected by Variable Expansion (t3 and t3_2)*** table, we see that the length of the new loop body is 4, greater than the lifetimes of the values in the loop.

Table 3-12: Modulo Schedule Corrected by Variable Expansion: t3 and t3_2

	Iteration 1	*Iteration 2*	*Iteration 3*	*Iteration 4 ...*
1	t1=[p1++]			
2	t2=[p2+ +],t3=t1+t5			
3	t4=t2+1,t5=t1+t3	t1=[p1++]		
4	t6=t4*t5	t2=[p2+ +],t3_2=t1+t5		
5	***t7=t6*t3***	***t4=t2+1,t5=t1+t3 _2***	***t1=[p1++]***	
6	***[p8++]=t7***	***t6=t4*t5***	***t2=[p2+ +],t3=t1+t5***	
7		***t7=t6*t3_2***	***t4=t2+1,t5=t1+t3***	***t1=[p1++]***
8		***[p8++]=t7***	***t6=t4*t5***	***t2=[p2+ +],t3_2=t1+t5***
9			t7=t6*t3	t4=t2+1,t5=t1+t3 _2
10			[p8++]=t7	t6=t4*t5
11				t7=t6*t3_2
12				[p8++]=t7

So the loop becomes:

```
t1=[p1++];

t2=[p2++],t3=t1+t5;
t4=t2+1, t5=t1+t3, t1=[p1++];
t6=t4*t5,          t2=[p2++],t3_2=t1+t5;
LOOP (N-2)/2
t7=t6*t3,          t4=t2+1,t5=t1+t3_2,  t1=[p1++];
[p8++]=t7,         t6=t4*t5,            t2=[p2++],t3=t1+t5;
                   t7=t6*t3_2,          t4=t2+1,t5=t1+t3,   t1=[p1++];
                   [p8++]=t7,           t6=t4*t5, t2=[p2++],t3_2=t1+t5;
END LOOP
                   t7=t6*t3,            t4=t2+1,t5=t1+t3_2;
```

```
                                             [p8++]=t7,          t6=t4*t5;
                                                                 t7=t6*t3_2;
                                                                 [p8++]=t7;
```

This process of duplicating the kernel and renaming colliding variables is called *variable expansion*, and the number of times the compiler duplicates the kernel is referred to as the *modulo variable expansion factor (MVE)*.

Conceptually we use different set of names, "register sets", for successive iterations of the original loop in progress in the unrolled kernel (in practice we rename just the conflicting variables, see the ***Instructions after Modulo Variable Expansion*** table). In terms of reading the code, this means that a single iteration of the loop generated by the compiler will be processing more than one iteration of the original loop. Also, the compiler will be using more registers to allow the iterations of the original loop to overlap without clobbering the live values.

In terms of stages:

```
SC0                                 // prolog
SC1    SC0_2                        // prolog
LOOP (N-2)/2              // That is N-(SC-1)/MVE, where SC=3, MVE=2
SC2    SC1_2    SC0                 // kernel
       SC2_2    SC1    SC0_2        // kernel
END LOOP
                SC2    SC1_2        // epilog
                       SC2_2        // epilog
```

where SCN_2 is SCN subject to renaming; in our case only occurrences of `t3` are renamed as `t3_2` in SCN_2.

In terms of instructions:

```
I1;                                             // prolog
I2,I3;                                          // prolog
I4,I5,     I1_2;                                // prolog
I6,        I2_2,I3_2;                           // prolog
LOOP(N-2)/2     // That is N-(SC-1)/MVE, where SC=3, MVE=2
I7,        I4_2,I5_2,  I1;                      // kernel
I8,        I6_2,       I2,I3;                   // kernel
           I7_2,       I4,I5,     I1_2;         // kernel
           I8_2,       I6,        I2_2,I3_2;    // kernel
END LOOP
                       I7,        I4_2,I5_2;    // epilog
                       I8,        I6_2;         // epilog
                                  I7_2;         // epilog
                                  I8_2;         // epilog
```

where `IN_2` is `IN` subject to renaming, in our case only occurrences of `t3` are renamed as `t3_2` in all `IN_2`, as seen in the ***Instructions after Modulo Variable Expansion*** table.

Table 3-13: Instructions after Modulo Variable Expansion

Generic Instruction	*Specific Instance*
	`t1=[p1++]`

Table 3-13: Instructions after Modulo Variable Expansion (Continued)

Generic Instruction	*Specific Instance*
`I1` and `I1_2`	
`I2` and `I2_2`	`t2=[p2++]`
`I3`	`t3=t1+t5`
`I3_2`	`t3_2=t1+t5`
`I4` and `I4_2`	`t4=t2+1`
`I5`	`t5=t1+t3`
`I5_2`	`t5=t1+t3_2`
`I6` and `I6_2`	`t6=t4*t5`
`I7`	`t7=t6*t3`
`I7_2`	`t7=t6*t3_2`
`I8` and `I8_2`	`[p8++]=t7`

Trip Count

Notice that as the modulo scheduler expands the loop kernel to add in the extra variable sets, the iteration count of the generated loop changes from `(N-SC)` to `(N-SC)/MVE`. This is because each iteration of the generated loop is now doing more than one iteration of the original loop, so fewer generated iterations are required.

However, this also relies on the compiler knowing that it can divide the loop count in this manner. For example, if the compiler produces a loop with `MVE=2` so that the count should be `(N-SC)/2`, an odd value of `(N-SC)` causes problems. In these cases, the compiler generates additional "peeled" iterations of the original loop to handle the remaining iteration. As with rotation, if the compiler cannot determine the value of `N`, it will make parts of the loop-the kernel or peeled iterations conditional so that they are executed only for the appropriate values of `N`.

The number of times the generated loop iterates is called the "trip count". As explained above, sometimes knowing the trip count is important for efficient scheduling. However, the trip count is not always available. Lacking it, additional information may be inferred, or passed to the compiler through the `loop_count` pragma, specifying:

- *Trip minimum*: a lower bound for the trip count
- *Trip maximum*: an upper bound for the trip count
- *Trip modulo*: a number known to divide the trip count

A Worked Example

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

```
#include <builtins.h>
float sp(float *a, float *b, int n) {
   int i;
```

```
   float sum=0;
   aligned(a, 2 * sizeof(float));
   aligned(b, 2 * sizeof(float));
   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 resembles 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 SHARC processors supported by the compiler have two compute units that may perform computations simultaneously. 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. On some SHARC processors, 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-word aligned. This is done in the above example by use of `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 times, 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 modulo-schedules 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 to generate the resultant assembly code. The decisions taken by the compiler as to whether certain optimizations are safe or worthwhile are generally invisible to a programmer. However, it can be beneficial to get feedback from the compiler

regarding the decisions made during optimization. The intention of the information provided is to give a programmer an understanding of how close to optimal a program is and what more could possibly be done to improve the generated code.

The feedback from the compiler optimizer is provided by means of annotations made to the assembly file generated by the compiler. The assembly file generated by the compiler can be kept by specifying the `-S` switch, the `-save-temps` switch, or by checking ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Save temporary files*** option in the IDE.

The assembly code generated by the compiler optimizer is annotated with the following information:

- Global Information
- Procedure Statistics
- Instruction Annotations
- Loop Identification
- Vectorization
- Modulo Scheduling Information
- Warnings, Failure Messages and Advice

The assembly annotations provide information in several areas that you can use to assist the compiler's evaluation of your source code. In turn, this improves the generated code. For example, annotations could provide indications of resource usage or the absence of a particular optimization from the resultant code. Annotations which note the absence of optimization can often be more important than those noting its presence. Assembly code annotations give the programmer insight into why the compiler enables and disables certain optimizations for a specific code sequence.

The assembly output for the examples in this chapter may differ based on optimization flags and the version of the compiler. As a result, you may not be able to reproduce these results exactly.

Annotation Examples

Your installation directory contains a number of examples which demonstrate annotation output of the optimizer. You can find these examples in the following directory tree:

<*installation*>`\SHARC\Examples\No_HW_Required\`*processor*`\annotations`

where *processor* is:

- `ADSP-21469` - contains IDE projects pre-configured for the ADSP-21469 processor.

Examples in this directory tree are not intended to be functional. Although they can be built in the IDE and loaded into a processor, they do not *do* anything of significance. Instead, their purpose is to show the annotations generated by the compiler, for a given input source code. In each case, you can import and build the example, as described in Importing Annotation Examples, then examine the resulting assembly file.

Depending on the example, you can also see annotations when viewing the C source file in the IDE. Details on how to view the generated annotations is given in:

- Viewing Annotation Examples in the IDE
- Viewing Annotation Examples in Generated Assembly

Importing Annotation Examples

To import an annotation example into the IDE:

1. Choose ***File*** > ***Import***.

   The ***Import*** wizard appears.

2. On the ***Select*** page, select >***General*** > ***Existing Projects Into Workspace***.
3. Click ***Next***.

   The ***Import Projects*** page appears.

4. In ***Select root directory***, browse to the `..\SHARC\Examples\No_HW_Required\`*processor*`\annotations` directory in your CCES installation.
5. Select `annotations` directory and click ***OK***.

   The ***Import Projects*** page with the annotation projects appears. Projects that cannot be imported are grayed out.

6. Click ***Select All***, ***Deselect All***, or select individual projects.
7. Enable the ***Copy projects into Workspace*** option.

   Enabling this option gives you a working copy of the example to build.

8. Click ***Finish***.

Once the IDE loads a project, build the project. Building a project produces an executable file and assembly source files.

The ***Console*** view shows any diagnostics resulting in the building process. Such a result is normal because annotations are a form of diagnostic and emitted to the standard error output as well as the assembly file.

Viewing Annotation Examples in the IDE

To view the annotations in the IDE:

1. Build the example executable, as described in Importing Annotation Examples.

2. Hover the mouse pointer over the `i` information icons in the marker bar of the editor area to view annotations associated with the source lines. Annotations are a low-severity form of diagnostic, gathered when the application is built.
3. Alternatively, open the ***Problems*** view to view the annotations.

The annotation example produces the `i` information icons because the project enables annotations diagnostics. Examine the project and note that the ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Warning*** > ***Warning/annotation/remark control to Errors, warnings and annotations*** option is enabled.

Viewing Annotation Examples in Generated Assembly

To view annotations in the generated assembly file:

1. Open the annotations project and build it, if you have not already done so.
2. In the ***Project Explorer*** view, browse to:
   - The `..\Debug\src` directory if you built the project using the ***Debug*** configuration
   - The `..\Release\src` directory if you built the project using the ***Release*** configuration.

   You find several assembly (`.s`) files there.
3. Double-click the assembly file that corresponds to the example.

   ADDITIONAL INFORMATION: For example, in the `file_position` example, double-click `file_example.s`.

The IDE opens the assembly file in the editor area. You can see the annotations as comments within the generated assembly file

You can see the generated assembly files because the annotations projects are configured with the ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***General*** > ***Save temporary files*** enabled. Normally, this setting is disabled, and the compiler deletes the generated assembly file after the file is converted into an object file.

Global Information

For each compilation unit, the assembly output is annotated with:

- The time of the compilation
- The options used during that compilation.
- The architecture for which the file was compiled.
- The silicon revision used during the compilation
- A summary of the workarounds associated with the specified architecture and silicon revision. These workarounds are divided into:

    - Disabled: the workarounds that were not applied
    - Enabled: the workarounds that were applied during the compilation.
    - Always on: the workarounds that are always applied and that cannot be disabled, not even by using the `-si-revision none` compiler switch.
    - Never on: these are workarounds that are never applied and that cannot be enabled.

The `global_information` project is an example of this information. Build the project, then open the `hello.s` assembly file. You will see this information at the start of the file.

Procedure Statistics

For each function, the following is reported:

- Frame size: size of stack frame.
- Registers used. Since function calls tend to implicitly clobber registers, there are several sets:
    - The first set is composed of the scratch registers changed by the current function. This does not count the registers that are implicitly clobbered by the functions called from the current function.
    - The second set are the call-preserved registers changed by the current function. This does not count the registers that are implicitly clobbered by the functions called from the current function.
    - The third set are the registers clobbered by the inner function calls.
- Inlined Functions. If inlining happens, then the header of the caller function reports which functions were inlined inside it and where. Each inlined function is reported using the position of the inlined call. All the functions inlined inside the inlined function are reported as well, generating a tree of inlined calls. Each node, except the root, has the form:

  `file_name:line:column'function_name` where:

    - *`function_name`* is the name of the function inlined.
    - *`line`* is the line number of the call to *`function_name`*, in the source file.
    - *`column`* is the column number of the call to *`function_name`*, in the source file.
    - *`file_name`* is the name of the source file calling *`function_name`*.

The `procedure_statistics` annotation example illustrates this. You can view the annotations in the IDE either via the C source window or the generated assembly.

- In a C source window, the procedure information for each function can be viewed by hovering the mouse pointer over the "i" information icon in the gutter beside the first line of each function declaration-for example, beside `int foo(int in)`, in `procedure_statistics.c`.
- In an assembly source window, the procedure information can be viewed by scrolling down to the label that marks the start of each function-for example, just after the label `_foo:` in `procedure_statistics.s`.

The procedure_statistics_inlining demonstrates the annotations produced when a function inlines the contents of another function. Build the project in the Release configuration, and open Release\src\procedure_statistics_inlining.s. Observe how calls to functions f2() and f3() have been inlined into function f1(), and how the annotations at label _f1: report this.

Note that, if you build using the Debug configuration, you do not see the same annotations, as the optimizer is not enabled, so inlining does not happen.

Instruction Annotations

Sometimes the compiler annotates certain assembly instructions. It does so in order to point to possible inefficiencies in the original source code, or when the -annotate-loop-instr compiler switch is used to annotate the instructions related to modulo scheduled loops.

The format of an assembly line containing several instructions is changed. Instructions issued in parallel are no longer shown all on the same assembly line; each is shown on a separate assembly line, so that the instruction annotations can be placed after the corresponding instructions. Thus,

```
instruction_1, instruction_2, instruction_3;
```

is displayed as:

```
instruction_1,  // {annotations for instruction_1}
instruction_2,  // {annotations for instruction_2}
instruction_3;  // {annotations for instruction_3}
```

Example instruction_annotations demonstrates both these kinds of annotation. Build the example using the Release mode.

- When viewing instruction_annotations.c in the C source window, you can see that there is an annotation in the bad_mod() function to indicate that the division operation is emulated in software. You can also see that the optimizer modulo-scheduled the loop in the dotprod() function, but the individual instruction annotations are not available.
- When viewing instruction_annotations.s in the assembly source window, you can see the same annotations as for the C source window, but you can also see the additional information for each instruction within the loop in the dotprod() function.

Loop Identification

One useful annotation is loop identification-that is, showing the relationship between the source program loops and the generated assembly code. This is not easy due to the various loop optimizations. Some of the original loops may not be present, because they are unrolled. Other loops get merged, making it difficult to describe what has happened to them.

The assembly code generated by the compiler optimizer is annotated with the following loop information:

- Loop Identification Annotations
- File Position

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 latter annotation follows the jump back to the beginning of the loop. However, there are cases in which the code corresponding to a user loop cannot be entirely represented between 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.

  In cases where a loop has been totally deleted (because a source-level loop is never entered), the compiler issues the following annotation (see Warnings, Failure Messages and Advice):

  ```
  cc2296: {D} annotation: Loop structure removed due to dead code
  elimination.
  ```

  In cases where the loop control code surrounding a loop body has been removed (because the loop always iterates only once), the compiler issues the following annotation (see Warnings, Failure Messages and Advice):

  ```
  cc1974: {D} annotation: loop always iterates once - loop converted to
  linear code.
  ```

- 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 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, the compiler shows how many of that resource are used, how many are available and the percentage of utilization during the entire loop. Resources are shown in decreasing order of utilization. Note that 100% utilization means that the corresponding resource is used at its full capacity and represents a bottleneck for the loop.
  - ***Register usage.*** If the `-annotate-loop-instr` compilation switch is used, then the register usage table is shown. This table has one column for every register that is defined or used inside the loop. The

header of the table shows the names of the registers, written on the vertical, top down. The registers that are not accessed do not show up. The columns are grouped on data registers, pointer registers and all other registers. For every cycle in a loop (including stalls), there is a row in the array. The entry for a register has a '*' on that row if the register is either live or being defined at that cycle.

If the code executes in parallel (in a SIMD region), accessing a D register usually means accessing its corresponding shadow register in parallel. In these cases, the name of the register is prefixed with 2x. For instance, `2xr2`.

- ***Optimizations.*** 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 sections.

- Sometimes the compiler generates additional loops that may or may not be directly associated with the loops in the user program. Whenever possible, the compiler annotations try to show the relation between such compiler-generated loops and the original source code.

The `loop_identification` annotation example shows some of these annotations. Build the example using the Release configuration. The function `bar()` in file `loop_identification.c` contains two loops, written in such a way that the second loop will not be entered: when the first loop completes, the conditions of entry to the second loop are false. When the optimizer is enabled, the compiler can detect this through a process called "constant propagation", and can delete the second loop entirely.

- When viewing `loop_identification.c` in a C source window, "i" information icons appear in the gutter next to the lines containing the for and while keywords that introduce loops. For the first loop, trip count, estimated cycle count and resource usage is given, while for the second loop, the annotation reports that the loop is removed due to constant propagation.
- When viewing `loop_identification.s` in an assembly source window, an annotation appears following the "`_bar:`" label, reporting the removed loop. At other points in the function, annotations appear showing that the following code is part of the first loop, or part of the top level of the function (i.e. not in any loop).

File Position

When the compiler refers to a file position in an annotation, it does so using the file name, line number, and the column number in that file as:

```
"ExampleC.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 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>`.

Annotations example `file_position` demonstrates this. When built using the Release configuration, two levels of inlining occur in file `file_position.c`:

- When viewing `file_position.c` in a C source window, the loop at the start of function `f3()` has an "i" information indicating that the loop has been inlined into function `f2()` twice, and that each of those instances have in turn been inlined into function function `f1()`.
- When viewing `file_position.s` in an assembly source window, annotations appear in the generated file immediately before the code for the loop. The annotations in function `f2()` indicate that the following code was inlined from function `f3()`, and the annotations in function `f1()` indicate that the following code was inlined from function `f2()`, which in turn was inlined from function `f3()`. There are also annotations at the start of functions `f2()` and `f1()` reporting which functions have been inlined into them, as described in Procedure Statistics.

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 from consecutive iterations of a loop and 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 500. The vectorization factor is the number of operations in the original loop that are executed at once in the transformed loop. It is illustrated using some pseudo code below.

```
for (i=0; i < 1000; i+=2)
   (a[i], a[i+1]) = (b[i],b[i+1]) .plus2. (c[i], c[i+1])
```

In the above example, the vectorization factor is 2.

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.

Unroll and Jam

A vectorization-related transformation is *unroll and jam*. Where the source file has two nested loops, sometimes the compiler can unroll the outer loop, to create two copies of the inner loop each operating on different iterations of

the loop. It can then "jam" these two loops together, interleaving their operations, giving a sequence of operations that is more amenable to vectorization. The compiler issues annotations when this transformation has happened.

The `unroll_and_jam` annotation example demonstrates this. The example contains three source files:

- `unroll_and_jam_original.c` - the "real" example. This file contains a function which the compiler is able to optimize using the unroll-and-jam transformation.
- `unroll_and_jam_unrolled.c` - this file is illustrative of how the compiler's internal representation would be, part-way through the unroll-and-jam transformation. This is *not* an example of how you should write your code. In this representation, the compiler has unrolled the outer loop once, so that there are two complete, separate copies of the inner loop. The first copy works on even iterations, while the second works on odd iterations.
- `unroll_and_jam_jammed.c` - another illustrative representation of the function, after the transformation is complete. The compiler has taken the two copies of the loop and overlapped them, then vectorized the operations so that the narrow loads and stores are now wide loads and stores that access two adjacent locations in parallel, and the accumulation operations do two separate additions in the same cycle.

NOTE: You should always write your code in the cleanest manner possible, to most clearly express your intention to the compiler. You should not attempt to apply transformations such as unroll-and-jam explicitly within your code, as that will obscure your intent and inhibit the optimizer. The unrolled and jammed files are only presented here to illustrate the behavior of the transformation.

The unroll_and_jam annotation example makes use of the `unroll_and_jam_original.c` file to demonstrate the annotation produced during this transformation. Build the example using the Release configuration.

- When viewing the `unroll_and_jam_original.c` file in a C source window, there is an "i" information icon next to the outer loop, reporting that the loop has been unrolled and jammed.
- When viewing the `unroll_and_jam_original.s` file in an assembly source window, there is an annotation preceding the generated code for the outer loop, reporting that the loop has been unrolled and jammed.

Loop Flattening

Another transformation, 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.

The `loop_flattening` annotation example demonstrates this. It contains two files to illustrate the transformation:

- `loop_flattening_original.c` -This file contains two nested loops, iterating 30 times and 100 times, respectively.
- `loop_flattening_flattened.c` - This file contains a single loop, iterating 3000 times. This file is *not* an example of how you should write your code, it is merely an illustration of the transformation applied by the compiler optimizer.

The `loop_flattening` annotation example uses `loop_flattening_original.c` to demonstrate the annotations produced. Build the example using the Release configuration.

- When viewing `loop_flattening_original.c` in a C source window, there is an annotation on the outer loop, indicating that the two loops were flattened into one.
- When viewing `loop_flattening_original.s` in an assembly source window, there is an annotation at the beginning of the function, indicating that the two loops were flattened into one; the annotation appears at the start of the function because a loop was "lost" (the loop's structure was removed), and lost loops are reported at the start of each function.

Vectorization Annotations

For every loop that is vectorized, the following information is provided:

- The vectorization factor
- The number of peeled iterations
- The position of the peeled iterations (front or back of the loop)
- Information about whether peeled iterations are conditionally or unconditionally executed

For every loop pair subject to loop flattening, the following information is provided:

- The loop that is lost
- The remaining loop that it was merged with

The `vectorization` annotation example demonstrates some of this. File `vectorization.c` contains a function `copy()` which the compiler can conditionally vectorize, when optimizing. Build the example using the Release configuration.

- When viewing `vectorization.c` in a C source window, there are "i" information icons next to the loop constructs in the `copy()` function. These annotations report that there are multiple versions of the loop, one of which is unvectorized; that a loop was vectorized by a factor of two; the trip counts for the loops; and so on.
- When viewing `vectorization.s` in an assembly source window, there are multiple versions of the loop in the function. One has annotations to indicate it has been vectorized, while the other has an annotation to indicate that it is the unvectorized version of the same loop.

Modulo Scheduling Information

For every modulo scheduled loop (see also Modulo Scheduling), in addition to regular loop annotations, the following information is provided:

- The initiation interval (II)
- The final trip count if it is known: the trip count of the loop as it ends up in the assembly code
- A cycle count representing the time to run one iteration of the pipelined loop

- The minimum trip count, if it is known and the trip count is unknown
- The maximum trip count, if it is known and the trip count is unknown
- The trip modulo, if it is known and the trip count is unknown
- The stage count (iterations in parallel)
- The MVE unroll factor
- The resource usage
- The minimum initiation interval due to resources (`res MII`)
- The minimum initiation interval due to dependency cycles (`rec MII`)

Annotations for Modulo Scheduled Instructions

The `-annotate-loop-instr` compiler switch can be used to produce additional annotation information for the instructions that belong to the prolog, kernel or epilog of the modulo scheduled loop.

Consider the example whose schedule is in the ***Modulo Schedule Corrected by Variable Expansion (t3 and t3_2)*** table (see Variable Expansion and MVE Unroll). Remember that this does not use a real DSP-architecture, but rather a theoretical one able to schedule four instructions on a line, and each line takes one cycle to execute. We can view the instructions involved in modulo scheduling as in the ***Modulo Scheduled Instructions*** table.

Table 3-14: Modulo Scheduled Instructions

	Part	*Iteration 0*	*Iteration 1*	*Iteration 2*	*Iteration 3 ...*
		Register Set 0	Register Set 1	Register Set 0	Register Set 1
1	prolog	I1			
2	prolog	I2, I3			
3	prolog	I4, I5	I1_2		
4	prolog	I6	I2_2, I3_2		
5		L: Loop ...			
6	*kernel*	*I7*	*I4_2, I5_2*	*I1*	
7	*kernel*	*I8*	*I6_2*	*I2, I3*	
8	*kernel*		*I7_2*	*I4, I5*	*I1_2*
9	*kernel*		*I8_2*	*I6*	*I2_2, I3_2*
10		END Loop			
11	epilog			I7	I4_2, I5_2
12	epilog			I8	I6_2
13	epilog				I7_2

Table 3-14: Modulo Scheduled Instructions (Continued)

	Part	*Iteration 0*	*Iteration 1*	*Iteration 2*	*Iteration 3 ...*
		Register Set 0	Register Set 1	Register Set 0	Register Set 1
14	epilog				I8_2

Due to variable expansion, the body of the modulo scheduled loop contains MVE=2 unrolled instances of the kernel, and the loop body contains instructions from 4 iterations of the original loop. The iterations in progress in the kernel are shown in the table heading, starting with `Iteration 0` which is the oldest iteration in progress (in its final stage). This example uses two register sets, shown in the table heading.

The instruction annotations contain the following information:

- The part of the modulo scheduled loop (prolog, kernel or epilog)
- The loop label. This is required since prolog and epilog instructions appear outside of the loop body and are subject to being scheduled with other instructions.
- ID: a unique number associated with the original instruction in the unscheduled loop that generates the current instruction. It is useful because a single instruction in the original loop can expand into multiple instructions in a modulo scheduled loop. In our example the annotations for all instances of `I1` and `I1_2` have the same id, meaning they all originate from the same instruction `(I1)` in the unscheduled loop.

  The IDs are assigned in the order the instructions appear in the kernel and they might repeat for MVE unroll > 1.
- Loop-carry path, if any. If an instruction belongs to the loop-carry path, its annotation will contain a `*'. If several such paths exist, `*2' is used for the second one, `*3' for the third one, etc.
- `sn`: the stage count the instruction belongs to.
- `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:
  - Iteration. `iter`: specifies the iteration of the original loop an instruction is on in the schedule.
  - In a modulo scheduled kernel, there are instructions from (`SC+MVE-1`) iterations of the original loop. `iter=0` denotes instructions from the earliest iteration of the original loop, with higher numbers denoting later iterations.

Thus, the instructions corresponding to the schedule in the ***Modulo Scheduled Instructions*** table for a hypothetical machine are annotated as follows:

```
1 : I1;          // {L10 prolog:id=1,sn=0,rs=0}
2 : I2,          // {L10 prolog:id=2,sn=0,rs=0}
3 :     I3;      // {L10 prolog:id=3,sn=0,rs=0}
4 : I4,          // {L10 prolog:id=4,sn=1,rs=0}
5 :     I5,      // {L10 prolog:id=5,sn=1,rs=0}
```

```
6 :     I1_2;    // {L10 prolog:id=1,sn=0,rs=1}
7 : I6,          // {L10 prolog:id=6,sn=1,rs=0}
8 :     I2_2,    // {L10 prolog:id=2,sn=0,rs=1}
9 :  I3_2;       // {L10 prolog:id=3,sn=0,rs=1}
10: //--------------------------------------------------------
11: //   Loop at ...
12: //--------------------------------------------------------
13: //   This loop executes 2 iterations of the original loop in estimated 4
cycles.
14: //--------------------------------------------------------
15: //   Unknown Trip Count
16: //   Successfully found modulo schedule with:
17: //     Initiation Interval (II)       = 2
18: //     Stage Count (SC)               = 3
19: //     MVE Unroll Factor              = 2
20: //     Minimum initiation interval due to recurrences
          (rec MII)                       = 2
21: //     Minimum initiation interval due to resources
          (res MII)                       = 2.00
22: //----------------------------------------------------------
23:L10:
23:LOOP (N-2)/2;
25: I7,           // {kernel:id=7,sn=2,rs=0,iter=0}
26:     I4_2,     // {kernel:id=4,sn=1,rs=1,iter=1}
27:     I5_2,     // {kernel:id=5,sn=1,rs=1,iter=1,*}
28:     I1;       // {kernel:id=1,sn=0,rs=0,iter=2}
29: I8,           // {kernel:id=8,sn=2,rs=0,iter=0}
30:     I6_2,     // {kernel:id=6,sn=1,rs=1,iter=1}
31:     I2,       // {kernel:id=2,sn=0,rs=0,iter=2}
32:     I3;       // {kernel:id=3,sn=0,rs=0,iter=2,*}
33: I7_2,         // {kernel:id=7,sn=2,rs=1,iter=1}
34:     I4,       // {kernel:id=4,sn=1,rs=0,iter=2}
35:     I5,       // {kernel:id=5,sn=1,rs=0,iter=2,*}
36:     I1_2;     // {kernel:id=1,sn=0,rs=1,iter=3}
37: I8_2,         // {kernel:id=8,sn=2,rs=1,iter=1}
38:     I6,       // {kernel:id=6,sn=1,rs=0,iter=2}
39:     I2_2,     // {kernel:id=2,sn=0,rs=1,iter=3}
40:     I3_2;     // {kernel:id=3,sn=0,rs=1,iter=3,*}
41:END LOOP
42:
43: I7,           // {L10 epilog:id=7,sn=2,rs=0}
44:     I4_2,     // {L10 epilog:id=4,sn=1,rs=1}
45:     I5_2;     // {L10 epilog:id=5,sn=1,rs=1}
46: I8,           // {L10 epilog:id=8,sn=2,rs=0}
47:     I6_2;     // {L10 epilog:id=6,sn=1,rs=1}
48: I7_2;         // {L10 epilog:id=7,sn=2,rs=1}
49: I8_2;         // {L10 epilog:id=8,sn=2,rs=1}
```

Lines 10-22 define the kernel information: loop name and modulo schedule parameters: II, stage count, and so on.

Lines 25-40 show the kernel.

Each instruction in the kernel has an annotation between {}, inside a comment following the instruction. If several instructions are executed in parallel, each gets its own annotation.

For instance, line 27 is:

```
27:      I5_2,    // {kernel:id=5,sn=1,rs=1,iter=1,*}
```

This annotation indicates:

- That this instruction belongs to the kernel of the loop starting at L10.
- That this and the other three instructions that have ID=5 originate from the same original instruction in the unscheduled loop:

```
5 :      I5,      // {L10 prolog:id=5,sn=1,rs=0}
...
27:      I5_2,    // {kernel:id=5,sn=1,rs=1,iter=1,*}
...
35:      I5,      // {kernel:id=5,sn=1,rs=0,iter=2,*}
...
45:      I5_2;    // {L10 epilog:id=5,sn=1,rs=1}
```

- sn=1 shows that this instruction belongs to stage count 1.
- rs=1 shows that this instruction uses register set 1.
- iter=1 specifies that this instruction belongs to the second iteration of the original loop (Iter numbers are zero-based).
- The `*' indicates that this is part of a loop carry path for the loop. In the original, unscheduled loop, that path is I5 -> I3 -> I5. Due to unrolling, in the scheduled loop the "unrolled" path is I5_2 -> I3 -> I5 -> I3_2 -> I5_2.

The prolog and epilog are not clearly delimited in blocks by themselves, but their corresponding instructions are annotated like the ones in the kernel except that they do not have an iter field and that they are preceded by a tag specifying to which loop prolog or epilog they belong.

Note that the prolog/epilog instructions may mix with other instructions on the same line. This situation does not occur in this example; however, in a different example it might have:

```
I5_2,    // {L10 epilog:id=5,sn=1,rs=1}
I20;
```

This shows a line with two instructions. The second instruction I20 is unrelated to modulo scheduling, and therefore it has no annotation.

Warnings, Failure Messages and Advice

Some programming constructs that have a negative effect on performance. Since you may not be aware of the hidden costs, the compiler annotations try to give warnings when such situations occur. Also, if a program construct

keeps the compiler from performing a certain optimization, the compiler gives the reason why that optimization was precluded.

In some cases, the compiler believes it could do a better job if you changed your code in certain ways. In these cases, the compiler offers advice on potentially beneficial code changes. However, take this cautiously. While it is likely that making the suggested change will improve the performance, there is no guarantee that it will actually do so.

Some of the messages are:

- ***This loop was not modulo scheduled because it was optimized for space***

  When a loop is modulo scheduled, it often produces code that has to precede the scheduled loop (the prolog) and follow the scheduled loop (the epilog). This almost always increases the size of the code. That is why, if you specify an optimization that minimizes the space requirements, the compiler doesn't attempt modulo scheduling of a loop.

- ***This loop was not modulo scheduled because it contains calls or volatile operations***

  Due to the restrictions imposed by calls and volatile memory accesses, the compiler does not try to modulo schedule loops containing such instructions.

- ***This loop was not modulo scheduled because it contains too many instructions***

  The compiler does not try to modulo schedule loops that contain many instructions, because the potential for gain is not worth the increased compilation time.

- ***This loop was not modulo scheduled because it contains jump instructions***

  Only single block loops are modulo scheduled. You can attempt to restructure your code and use single block loops.

- ***This loop would vectorize if alignment were known***

  The loop was not vectorized because of unknown pointer alignment.

- ***Consider using pragma loop_count to specify the trip count or trip modulo***

  This information may help vectorization.

- ***Consider using pragma loop_count to specify the trip count or trip modulo, in order to prevent peeling***

  When a loop is vectorized, but the trip count is not known, some iterations are peeled from the loop and executed conditionally (based on the run-time value of the trip count). This can be avoided if the trip count is known to be divisible by the number of iterations executed in parallel as a result of vectorization.

- ***Operation of this size is implemented as a library call***

  This message is issued when source code operator *operation* results in a library call, due to lack of hardware support for performing that operation on operands of that size. In this case the compiler also issues the following remark (see Warnings, Failure Messages and Advice):

```
cc2302: {D} annotation: operation of this size is implemented as a
library call.
```

- *Operation is implemented as a library call*

  This message is issued when source code operator *operation* results in a library call, due to lack of direct hardware support. For instance, an integer division results in a library call. In this case the compiler also issues the following remark (see Warnings, Failure Messages and Advice):

  ```
  cc2302: {D} annotation: operation is implemented as a library call.
  ```

- *MIN operation could not be generated because of unsigned operands*

  This message is issued when the compiler detects a MIN operation performed between unsigned values. Such an operation cannot be implemented using the hardware MIN instruction, which requires signed values.

- *MAX operation could not be generated because of unsigned operands* This message is issued when the compiler detects a MAX operation performed between unsigned values. Such an operation cannot be implemented using the hardware MAX instruction, which requires signed values.

- *Use of volatile in loops precludes optimizations*

  In general, volatile variables hinder optimizations. They cannot be promoted to registers, because each access to a volatile variable requires accessing the corresponding memory location. The negative effect on performance is amplified if volatile variables are used inside loops. However, there are legitimate cases when you have to use a volatile variable exactly because of this special treatment by the optimizer. One example would be a loop polling if a certain asynchronous condition occurs. This message does not discourage the use of volatile variables, it just stresses the implications of such a decision.

- *Jumps out of this loop prevent efficient hardware loop generation*

  Due to the presence of jumps out of a loop, the compiler either cannot generate a hardware loop, or was forced to generate one that has a conditional exit.

- *There are N more instructions related to this call*

  Certain operations are implemented as library calls. In those cases the call instruction in the assembly code is annotated explaining that the user operation was implemented as a call. However the cost of the operation may be slightly larger than the cost of the call itself, due to additional overhead required to pass the parameters and to obtain the result. This message gives an estimate of the number of instructions in such an overhead associated with a library call.

- *This function calls the "alloca" function which may increase the frame size*

  The assembly annotations try to estimate the frame size for a given function. However, if the function makes explicit use of `alloca` then this increases the frame size beyond the original reported estimate.

Analyzing Your Application

The compiler and run-time libraries provide several features for analyzing the run-time behavior of your application. These features allow you to better debug errors and fine tune the program. Features discussed in this chapter are:

- Application Analysis Configuration discusses how the profiles, log files and reports are named.
- Profiling With Instrumented Code discusses how to profile the application, measuring the time spent in individual functions in an application.
- Profile-Guided Optimization and Code Coverage discusses how to improve application performance using profile-guided optimization. Producing code coverage reports using profile-guided optimization data is also discussed.
- Heap Debugging details how to use the run-time library heap debugging feature to identify heap-allocated memory leaks and heap-allocated memory corruption within an application.
- Stack Overflow Detection details how to use the stack overflow feature to determine when an application has exceeded its maximum stack size.

Application Analysis Configuration

The analysis features described in this section can be configured through some global settings which are used by an underlying profiling layer. This layer is exposed by the `<sys/adi_prof.h>` header file. The following aspects can be controlled through this layer:

- Application Analysis and File Naming
- Device for Profiling Output
- Frequency of Flushing Profile Data

Application Analysis and File Naming

The analysis features described in this section each rely on files created by the application while it is running. In order for the analysis tools to be able to locate such files, the application and the tools must agree on the files' names. This is achieved through the use of the linker's `EXECUTABLE_NAME` directive, which allows an application to discover the name of its own executable image. The run-time library can then use this name as the basis of the generated files, thereby tying the generated file to the executable that created it. This allows the Reporter Tool to produce useful reports based on the application and its generated log files.

The features that make use of this functionality are:

- Profile-guided optimization (PGO) for hardware (see Using Profile-Guided Optimization With Hardware)
- Instrumented profiling (see Generating an Application With Instrumented Profiling)
- Heap debugging (see Heap Debugging)

The EXECUTABLE_NAME directive takes an assembler symbol name as a parameter. For the features in this chapter, the symbol name must be __executable_name on processors without support for byte-addressing, or _executable_name. on processors that support byte-addressing.

NOTE: You do not need to add the __executable_name or _executable_name. symbol to your application. The linker will automatically create an object file containing the declaration of the symbol when it encounters the EXECUTABLE_NAME directive in the .ldf file.

The __executable_name or _executable_name. assembler symbol declared in the .ldf file can be referenced in C/C++ applications. The data is stored in a NUL-terminated C string.

As an alternative to using the EXECUTABLE_NAME directive, you can provide a declaration of the symbol within your application, for example, in C:

```
char _executable_name[] = "my_executable.dxe";
```

On processors that support byte-addressing, this definition must be contained in a source file that is built using the -char-size-8 switch.

NOTE: If no __executable_name/_executable_name. symbol is defined in your application and no EXECUTABLE_NAME directive is provided in the .ldf file, the application will revert to using the default definition. This contains the string unknown.dxe.

Device for Profiling Output

The profiling features require an underlying I/O device driver to produce output to either stderr or the appropriate log file. The features will use the device driver specified by the integer adi_prof_io_device. If adi_prof_io_device is -1, the profilers will use the default device driver. adi_prof_io_device defaults to -1, but this definition can be overridden with a value representing the required device driver.

Frequency of Flushing Profile Data

To reduce the impact of I/O operations, the profilers buffer data internally, and write the data to the log files in bursts. The intervals can be controlled through the following global variables:

- adi_prof_min_flush_interval determines the minimum time that must pass between buffer flushes.
- adi_prof_max_flush_interval determines the maximum time that may pass between buffer flushes. This value is used to determine whether to flush data to the log file before the buffer fills.

The library provides default values for each of these variables, but you can override the defaults just by defining your own versions. For example:

```
uint32_t adi_prof_max_flush_interval =
          ADI_MSEC_FLUSH_INTERVAL(10000); // 10 seconds
```

NOTE: The ADI_MSEC_FLUSH_INTERVAL macro is based upon the __PROCESSOR_SPEED__ macro.

Profiling With Instrumented Code

Instrumented profiling is an application profiling tool that provides a summary of cycle counts for functions within an application.

Instrumented profiling works by planting function calls into your application which record the cycle count (and in multi-threaded cases, the thread identifier) at certain points. Applications built with instrumented profiling should be used for development and not be released.

Instrumented profiling requires that an I/O device is available in the application to produce its profiling data. The default I/O device will be used to perform I/O operations for instrumented profiling.

Instrumented profiling flushes any remaining profile data still pending when `exit()` is invoked. Multi-threaded applications may need to flush data explicitly.

To produce an instrumented profiling summary, perform these steps:

1. Compile your application with the `-p` switch or ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor*** > ***Enable compiler instrumented profiling*** selected. For best results use the optimization switches that will be enabled in the released version of the application.
2. Gather the profile. Run the executable with a training data set. The training data set should be representative of the data that you expect the application to process in the field. The profile is stored in a file with the extension `>.prf`.
3. Generate the profiling report. Two options for creating reports are available:
   a. Using the IDE; this produces an HTML format report.
   b. Using the command line tools; this will produce a plain-text report.
4. Based on the profiling report, modify the application to improve performance in critical sections of code.

Generating an Application With Instrumented Profiling

The `-p` compiler switch enables instrumented profiling in the compiler when compiling C/C++ source into assembly. The compiler cannot instrument assembly files or files that have already been compiled into object files.

You can enable the `-p` switch in the IDE via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Processor*** > ***Enable compiler instrumented profiling***.

NOTE: When compiling with the `-p` switch, the compiler and linker define the preprocessor macro `_INSTRUMENTED_PROFILING` with a value of 1.

Running the Executable File

To produce a profiling report, run the application in the simulator or on hardware. The application produces a profiling file which is used to create the profiling report. The profiling file is located in the same directory as the executable, and named as per the executable with a `.prf` suffix.

NOTE: If the `.ldf` file of the application does not use the `EXECUTABLE_NAME` directive, the profiling file reverts to the legacy name of unknown `.prf`. For more information, see Application Analysis and File Naming.

Convert the profiling output file into a readable report using one of two available tools: the IDE reporter or the `instrprof.exe` command line. See Generating an Instrumented Profiling Report and Invoking the `instrprof.exe` Command-Line Reporter for information on how to produce a report from the `.prf` profile data file.

Generating an Instrumented Profiling Report

The default output format for the Instrumented Profiling report is HTML. To produce the HTML file:

1. Choose ***File*** > ***New*** > ***Code Analysis Report***.

   The ***Analysis Report*** wizard appears.

2. Select ***Instrumented profiling*** and click ***Next***.

   The ***Input/output files for code analysis report generation*** page appears.

3. In ***DXE that produced the data file***, enter the name of the application executable file. Alternately, browse to the file.

4. In ***Data file***, enter the name of the profiling (`.prf`) data file. Alternately, browse to the file.

   ADDITIONAL INFORMATION: If the application `.ldf` file does not contain an `EXECUTABLE_NAME` directive, the ***Data file*** filed is not populated automatically.

5. In ***Report document's output file name***, enter the name of the new HTML report file. Alternately, browse to the existing file.

6. Click ***Finish***.

To generate a report from the command line, in either pre-formatted HTML or unformatted XML, refer to the Reporter.exe Command Line Report Generation Utility.

Invoking the `instrprof.exe` Command-Line Reporter

The `instrprof.exe` command-line tool produces a plain-text report printed to the command-line console.

To produce a report:

1. Invoke the `instrprof.exe` tool, providing the names of the executable and `.prf` profiling data files of the application as parameters.

   For example: `instrprof.exe test.dxe test.prf`

Standard output displays the report in the console or command-line interface.

Contents of the Profiling Report

The profiling report lists each profiled function called in the application, how many times it was called, and cycle counts for that function. In multi-threaded applications, the thread identifier is also displayed. The Reporter Tool and instrprof command-line program present the same information, but in different formats according to their output media. The respective formats are described in Reporter Tool Report Format and The instrprof Command Line Tool Report Format.

Example Program for Instrumented Profiling

```
int apples, bananas;

void apple(void) {
    apples++;   // 10 cycles
}

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

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

In the program shown in the ***Example Program for Instrumented Profiling***, assume that `apple()` takes 10 cycles per call and assume that `banana()` takes 20 cycles per call, of which 10 are accounted for by its call to `apple()`. The program, when run, calls `apple()` three times: twice directly and once indirectly through `banana()`. The `apple()` function clocks up 30 cycles of execution, and this is reported for both its inclusive and exclusive times, since `apple()` does not call other functions. The `banana()` function is called only once. It reports 10 cycles for its exclusive time, and 20 cycles for its inclusive time. The exclusive cycles are for the time when `banana()` is incrementing bananas and is not waiting for another function to return, and so it reports 10 cycles. The inclusive cycles include these 10 exclusive cycles and also include the 10 cycles `apple()` used when called from `banana()`, giving a total of 20 inclusive cycles.The `main()` function is called only once, and calls three other functions (`apple()` twice, `banana()` once). Between them, `apple()` and `banana()` use up to 40 cycles, which appear in the `main()` function's inclusive cycles. The `main()` function's exclusive cycles are for the time when `main()` is running, but is not in the middle of a call to either `apple()` or `banana()`.

NOTE: Time spent in unprofiled functions will be added to the exclusive cycle count for the innermost profiled function, if one is active. An active profiled function is a profiled function that has an entry in the call stack; that is, it has begun execution but has not yet returned. For example, if `apple()` called the system function `malloc()`, the time spent in `malloc()` that is uninstrumented is added to the time for `apple()`.

Reporter Tool Report Format

The HTML-formatted instrumented profiling report, produced by the IDE's Reporter Tool, contains a summary of information for the application. Each profiled function called during execution is listed with the following information:

- The function's name.
- The pathname of the source file containing the function.
- The number of times this function was called.
- "Number of cycles without calls": the total number of cycles spent executing the code of this function; if the function calls other profiled functions, the cycles spent in those functions is not included in this figure. Note that if the function calls other *non*-profiled functions, this figure *will* include the cycles spent in those functions.
- "Number of cycles with calls": the total number of cycles spent executing this function, or any function it calls. In other words, this figure gives the sum of cycle counts between this function being called, and it returning.
- The percentage of time spent in this function. This percentage is based on the "number of cycles without calls."
- The thread identifier, for a multi-threaded application.

The instrprof Command Line Tool Report Format

The `instrprof.exe` tool emits a report to standard output. The following is an example of the instrprof output:

```
Summary for thread 1
Function Name       ExecCount       Fn Only       Fn+nested
         main               1            40              80
        apple               3            30              30
       banana               1            10              20
Functional Summary:
Function Name       ExecCount       Fn Only       Fn+nested
         main               1            40              80
        apple               3            30              30
       banana               1            10              20
```

This report includes the following information, for each profiled function:

- The function's name.
- `ExecCount`: the number of times this function was called.
- `Fn Only`: this is the same value as "Number of cycles without calls", as described in Reporter Tool Report Format.
- `Fn+nested`: this is the same value as "Number of cycles with calls", as described in Reporter Tool Report Format.

The report gives a breakdown for each thread in the application, plus an overall combined report for all threads. In this single-threaded example, there is only one thread, so both portions of the report contain the same information.

Profiling Data Storage

The profiling information is stored at runtime in memory allocated from the system heap. If the profiling run-time support cannot allocate from the heap (usually because the heap is exhausted), the profiling runtime calls `adi_fatal_error()` and stops execution. The profiling data available when this happens will be incomplete and probably not very useful. To avoid this problem, increase the size of the system heap until the error is no longer seen when running. For more information, see Controlling System Heap Size and Placement.

NOTE: Although instrumented profiling uses the default heap for some of its internal storage, none of these allocations will appear in a heap usage report.

Computing Cycle Counts

When profiling is enabled, the compiler instruments the generated code by inserting calls to a profiling library at the start and end of each compiled function. The profiling library samples the processor's cycle counter and records this figure against the function just started or just completed. The profiling library itself consumes some cycles, and these overheads are not included in the figures reported for each function. Therefore, the total cycles reported for the application by the profiler are less than the cycles consumed during the life of the application.

In addition to this overhead, there is some approximation involved in sampling the cycle counter, because the profiler cannot guarantee how many cycles will pass between a function's first instruction and the sample. This is affected by the optimization levels, the state preserved by the function, and the contents of the processor's pipeline. The profiling library knows how long the call entry and exit takes on average, and adjusts its counts accordingly. Because of this adjustment, profiling using instrumented code provides an approximate figure, with a small margin for error. This margin is more significant for functions with a small number of instructions than for functions with a large number of instructions.

Multi-Threaded and Non-Terminating Applications

When an instrumented application is executed, it records data in the application, occasionally flushing this data to the host computer. In multi-threaded applications and non-terminating single-threaded applications, a request to flush data is required to ensure that all the profiling data is flushed from the application.

NOTE: In multi-threaded projects the default thread stack size may not be sufficient for profiling some applications, and may result in unexpected runtime behavior. Refer to your RTOS documentation for instructions on increasing your thread stack size.

Flushing Profile Data

To flush profiling data, the application needs to include the header file `instrprof.h` and call the function `instrprof_request_flush()`. Any changes to the code for instrumented profiling can be guarded by the preprocessor macro `_INSTRUMENTED_PROFILING`.

For example:

```
#if defined(_INSTRUMENTED_PROFILING)
   #include <instrprof.h>
#endif
void myfunc_noreturn(int x) {
   while ( 1 ) {
        // Perform operations
#if defined(_INSTRUMENTED_PROFILING)
   instrprof_request_flush();
#endif
    }
}
```

The flush occurs when the call to `instrprof_request_flush()` is made. Flushing cannot occur when the scheduler is disabled or from within interrupt handlers.

Profiling of Interrupts and Kernel Time

A single-threaded application (that is, one not built with the `-threads` compiler switch) adds any time spent in interrupts to the time of the innermost, active profiled function that was interrupted. Time spent in the interrupt handler is not visible in the profiling report produced. The compiler does not instrument functions declared as event handlers.

In a multi-threaded application using a real-time operating system (RTOS), only the time spent in the objects compiled with instrumentation is measured. Time spent in the scheduler/kernel and interrupt handlers is not reported. In the HTML- formatted report produced by the Reporter Tool, the ***percentage of time*** field is a percentage of the profiled time, not the absolute time that the application was running.

Behavior That Interferes With Instrumented Profiling

Several features of the C and C++ programming languages can have an impact on profiling results. The following features can result in unexpected results from profiling:

- ***Unexpected termination of application.*** If the application terminates unexpectedly, a complete set of profiling information may not be available. To ensure the profiling information is complete, all threads of execution should terminate by unwinding their stack (returning from `main()` or their thread creation function), or by calling `exit()`. RTOS-based systems may use a different implementation of `exit()`, so may require that data be flushed explicitly.
- ***Unexpected flow control.*** Functions that perform unexpected flow control, such as interrupts, C `setjmp/longjmp`, C++ exceptions, or calling other instrumented functions via `asm()` statements, may all result in inaccurate profiling information. Instrumented profiling relies on the typical C/C++ behavior of call and return to be able to measure cycle counts in functions. When features, such as `setjmp` or C++ exceptions, return through multiple stack frames, instrumented profiling will attempt to complete the profiling information for any stack frames unwound, but this may be inaccurate.

Profile-Guided Optimization and Code Coverage

The data recorded when running an application built with profile-guided optimization (see Using Profile-Guided Optimization) can also be used to generate a code coverage report using the IDE's Reporter Tool. A code coverage report provides a listing of your application's C/C++ source with execution counts for individual lines of code.

To produce a pre-formatted HTML code coverage report from the IDE, follow these steps:

1. Compile the application for profile-guided optimization for either simulators (see Using Profile-Guided Optimization With a Simulator) or hardware (see Using Profile-Guided Optimization With Hardware).
2. Run the application to produce a `.pgo` file.
3. Select ***File*** > ***New*** > ***Code Analysis Report***.
4. Ensure that ***Code coverage*** is selected.
5. Enter the name of the application executable in the ***DXE that produced the data*** field.
6. If the application `.ldf` file does not contain an `EXECUTABLE_NAME` directive, the ***Data file*** field will not have been automatically updated. Enter the name of the `.pgo` profiling data file into the field.
7. Enter the filename for the HTML report that will be generated.
8. Click ***Finish***.

To generate a report from the command line, in either pre-formatted HTML or unformatted XML, refer to the Reporter.exe Command Line Report Generation Utility.

The Code Coverage Report

The code coverage report contains a function-by-function summary of the application. For each C and C++ source file compiled with profile-guided optimization, a line count is displayed indicating how many times that line was executed.

Unexpected Line Counts in a Code Coverage Report

Several compiler features may impact the accuracy of a code coverage report. Compiler optimizations may rearrange code for better efficiency, and in some cases remove sections of code. This may result in unexpected line count information being displayed in the code coverage report.

If the application was compiled for profile-guided optimization on hardware, no line count information is reported for any function declared with an interrupt handler pragma.

If the `.pgo` file already exists when you run your application to gather a profile, the new profile data will accumulate into the same existing `.pgo` file rather than replacing it. This allows you to run your application under a number of different conditions and gather an overall coverage report.

Heap Debugging

The support for heaps provides convenient access to dynamic memory within an application. While this is an easy and efficient way to use dynamic memory, the lack of bounds checking associated with pointer accesses means that

mistakes are easy to make, and may have unpredictable side effects that are hard to identify and debug. CCES provides a heap debugging library that can be used to detect errors in the use of the heap, helping to identify issues that can cause unintended behavior.

The heap debugging library constrains debug versions of the heap manipulation functions (such as `malloc`, `free`, `new`, `delete`) provided by the C and C++ run-time libraries. These libraries record the heap activity and attempt to identify any potential issues with the usage of the heap, such as writing beyond the bounds of a buffer or failing to free memory.

The heap debugging library maintains a record of allocated blocks within the heap to track the current state of the heap. This recorded information is used as a reference to ensure that any heap allocations are valid; for example, making sure that the block being freed has been allocated by `calloc`, `malloc`, `realloc`, `new` or any derivatives and hasn't been freed previously. A guard region of 12 bytes, filled with a known bit pattern, is written before and after each block allocated from the heap and is checked at de-allocation to detect any overwriting of the bounds of the block. These bit patterns can be changed at build or run time to avoid the bit patterns corresponding to any application data that may be written into them, causing the bounds overflow to go undetected.

A cleanup function, `adi_heap_debug_end`, described in the *C/C++ Library Manual for SHARC Processors* detects any potential memory leaks (memory that has been allocated but not de-allocated) and heap corruption. This function is registered via `atexit`, and so is invoked if an application calls `exit` or returns from `main`.

The heap debugging library can generate a report detailing heap usage and errors via the Reporter Tool, to provide diagnostics via `stderr` at run-time, to check heap(s) for corruption, and to generate a current heap state snapshot of the heap(s).

Use the heap debugging library by linking it with your application. This means that source code does not need to be rebuilt. The heap debugging library also contains additional functions to allow the behavior of the heap debugging to be modified or, for additional diagnostic tests, to be carried out at runtime. These additional functions can be used by including the header `heap_debug.h` and requires your code to be rebuilt. For more information, see ***heap_debug.h*** in Chapter 1 of the *C/C++ Library Manual for SHARC Processors*.

The heap debugging library can be enabled in the IDE via ***Project > Properties > C/C++ Build > Settings > Tool Settings > Compiler > Run-time Checks > Link against heap debugging libraries***.

For a comprehensive list of errors detected by the heap debugging library, refer to ***Library Error Specific Codes*** in Chapter 1 of the *C/C++ Library Manual for SHARC Processors*.

Since the heap debugging library requires additional memory for code and data, an application may fail to link for projects that do not have sufficient additional memory. Heap and stack usage is also increased, so run-time errors may occur if insufficient stack or heap is available within your application.

The heap debugging library requires an underlying I/O device driver to produce output to either `stderr` or the `.hpl` file, as described in Device for Profiling Output.

Calls to heap allocation and de-allocation functions also take longer when heap debugging is enabled than if it is disabled, especially if report generation is enabled.

- Getting Started With Heap Debugging

- Using the Heap Debugging Library
- Generating a Heap Debugging Report

Getting Started With Heap Debugging

To use heap debugging, you first need to link your application against the heap debugging library instead of the normal heap library. You may also need to modify your application to perform some initial configuration, depending on:

- Whether your application is single- or multi-threaded
- The levels of logging and diagnostics you require

This section contains these subsections:

- Linking With the Heap Debugging Library. This section covers how to activate the heap debugging library.
- Heap Debugging Macro. This section explains how to conditionally include configuration code in your application.
- Default Behavior. This section describes how the out-of-the-box configurations for the heap debugging library.
- Additional Heap Overheads. This section gives a brief summary of the additional data requirements of heap debugging.
- Heap Debugging Report. This section identifies the file produced by the heap debugging library.

Linking With the Heap Debugging Library

Enable the heap debugging library:

1. In the IDE, select ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Run-time Checks*** > ***Link against heap debugging libraries***.
2. (Alternatively) On the command line, use the `-rtcheck-heap` switch.

Heap Debugging Macro

When ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Run-time Checks*** > ***Link against heap debugging libraries*** has been selected, the macro `_HEAP_DEBUG` is defined in the compiler, assembler and linker.

This macro is used in the header file `heap_debug.h` to define either prototype functions when enabled, or to use macros to replace any heap debugging specific function calls with statements mimicking a successful return from that function. This allows code to work independently of the heap debugging library being linked, with minimal performance overhead when the heap debugging library is not used. For more information, see ***heap_debug.h*** in Chapter 1 of the *C/C++ Library Manual for SHARC Processors*.

The `_HEAP_DEBUG` macro is also used to control the linking of the heap debugging library in the default `.ldf` files.

Default Behavior

The behavior of the heap debugging libraries can be configured either at build time or at runtime. The ***Default Configuration for Heap Debugging*** table shows the default configuration.

Table 3-15: Default Configuration for Heap Debugging

Generate `.hpl` log file	Enabled
Generate diagnostics to `stderr`	Disabled

The choice of configuration will affect the run-time performance of the application. For example, an application configured to log all heap activity to a file will make far more calls to the I/O library than an application configured only to emit an error diagnostic when a problem is encountered. However, the choice of configuration does not affect the additional code/data requirements imposed, as the heap debugging library has to record the same information in order to detect errors, regardless of whether that information is also being written to an activity log.

By default, applications generate an `.hpl` file of the heap activity; see Heap Debugging Report. The file can be converted into an HTML report for later analysis.

By default, no diagnostics regarding heap usage are written to `stderr`. You can enable `stderr` diagnostics by calling

```
adi_heap_debug_enable(_HEAP_STDERR_DIAG);
```

If your application does not terminate via `exit` or by returning from `main`, the heap debugging cannot track memory leaks or some cases of heap corruption. You must call `adi_heap_debug_end` at a suitable point in the application. Calling `adi_heap_debug_end` instructs the heap debugging library to check for any memory leaks and corruption before cleaning up any internal data used.

If `adi_heap_debug_end` is not called, either manually or via `exit`, then memory leaks can be identified in the report by the presence of a memory allocation without a corresponding de-allocation. Heap corruption can be detected by calling `adi_verify_all_heaps` from anywhere within your application.

For more information, refer to ***adi_verify_all_heaps*** in the *C/C++ Library Manual for SHARC Processors*.

Additional Heap Overheads

In addition to the over-allocation of each memory block by 24-bytes to use as a guard region around the block, the heap debugging library uses the system heap to allocate memory used for internal data. Approximately 24-bytes of memory is allocated from the system heap per allocation made from any heap, and 24-bytes of memory is allocated from the system heap to record information about each heap in the system.

Heap Debugging Report

The heap debugging library uses the symbol `__executable_name` or `_executable_name.`, provided by the `EXECUTABLE_NAME()` LDF instruction, to determine the name of the .hpl file used to generate the heap debugging report. If the symbol is not present then the file `unknown.hpl` is used. For more information, see Application Analysis and File Naming.

Using the Heap Debugging Library

The following sections describe the use of the heap debugging library. They detail the type of errors detected by the heap debugging library, explain how to generate a report detailing the heap usage and any errors from the `.hpl` file created by the heap debugging library, and explain how to enable `stderr` diagnostic reporting at runtime.

- Detected Errors. Lists the issues that the heap debugging library can detect.
- Viewing Reports. Explains how to convert the generated `.hpl` log file into report in HTML format.
- stderr Diagnostics. Covers how to control diagnostics emitted to the standard error stream.
- Call Stack. Discusses the call stack recorded with each heap operation, and how to configure it.
- Setting the Severity of Error Messages. Explains how to change the severity of each encountered issue.
- Default Diagnostic Severities. Lists the severity levels used by default.
- Guard Regions. Discusses the memory spaces allocated before and after each heap block, to detect writes beyond the block boundaries.
- Enabling and Disabling Features. Explains how to configure the library at built-time and at runtime.
- Buffering. Covers setting up a buffer to capture heap information while I/O is not possible.
- Pausing Heap Debugging. Explains how the tracing may be temporarily suspended.
- Finishing Heap Debugging. Provides advice on making sure heap tracing information is flushed to the log file.
- Verifying Heaps. Describes how to programmatically ensure that heaps are consistent.
- Behavior of Heap Debugging Library. Notes that using the heap debugging library has an impact on the characteristics of an application, compared to the normal heap library. It also describes these effects.
- Unfreed File I/O Buffers. Describes a side-effect of the inter-dependence between the heap library and I/O library.
- Memory Used by Operating Systems. Indicates that any heap usage by the RTOS may lead to additional entries in the heap log.

Detected Errors

The following errors are detected by the heap debugging library:

- Allocation of length zero
- Allocations which are bigger than the heap
- De-allocation of a previously de-allocated memory
- De-allocation of a pointer not returned by an allocation function
- `delete[]` of memory allocated by `new`
- `delete[]` of memory allocated by C functions (`calloc`, `malloc`, `realloc`)

- `delete` of memory allocated by C functions (`calloc`, `malloc`, `realloc`)
- `delete` of memory allocated by `new[]`
- `free` of memory allocated by C++ allocator operations
- `free` of null pointer
- `free` from incorrect heap
- Memory leaks (memory which has not been de-allocated)
- `realloc` of memory allocated by C++ allocator operations
- `realloc` of pointer not returned by allocation function
- `realloc` from incorrect heap
- Using heap functions from within an interrupt
- Writing beyond the scope of allocated memory block (up to 12 bytes before and after allocated memory)
- Writing to memory which has been de-allocated

Using the known bit patterns written in and around blocks by the heap debugging library can help to identify erroneous reads by the presence of these bit patterns in live data. These erroneous reads may be from:

- Memory that has been allocated from the heap but is uninitialized
- Memory that has been de-allocated
- Memory that is beyond the scope of the allocation (up to 12 bytes before or after allocated memory)

See Guard Regions for more information on these bit patterns.

Viewing Reports

To create an HTML report for your application's heap activity:

1. Build the application with ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Run-time Checks*** > ***Link against heap debugging libraries*** or `-rtcheck-heap`.
2. Run the application to produce a `.hpl` file.
3. Select ***File*** > ***New*** > ***Code Analysis Report***.
4. Ensure that ***Heap debugging*** is selected.
5. Enter the name of the application executable in the ***DXE that produced the data*** field.
6. If the application `.ldf` file does not contain an `EXECUTABLE _NAME` directive, the ***Data file*** field will not have been automatically updated. Enter the name of the `.hpl` profiling data file into the field.
7. Enter the filename for the HTML report that will be generated.
8. Click ***Finish***.

stderr Diagnostics

The heap debugging library can provide console diagnostic reporting for any issues detected with the heap usage, writing diagnostic messages to `stderr` as they are detected.

To enable `stderr` diagnostic reporting at runtime, call:

```
#include <heap_debug.h>
adi_heap_debug_enable(_HEAP_STDERR_DIAG);
```

To enable `stderr` diagnostics at build time, define the following C variable in your application source:

```
bool adi_heap_debug_stderr_diag = true;
```

The `stderr` diagnostics can have one of three severities: error, warning and ignored.

Errors print a diagnostic message and then call adi_fatal_error. For more information about `adi_fatal_error`, see the ***Fatal Error Handling*** section in the *C/C++ Library Manual for SHARC Processors*.

Warnings print a diagnostic message and then continue the application as normal. Ignored errors do not produce any diagnostic messages and do not terminate the application.

The severity of errors does not have any impact on the content of the generated heap debugging output file (`.hpl`), nor the heap debugging report generated from it. All errors are included.

Generated diagnostics are in the following form:

```
Heap [severity] in block [address]: [message]
```

when a memory address is relevant, or in the form:

```
Heap [severity]: [message]
```

when no memory address is relevant. Both are followed by a call stack where one is known and relevant. The `severity` is either `error` or `warning`; `address` represents the address of the memory block concerned, as returned by the allocation function. `message` is a short description of the issue that has been detected.

The call stack reported is the call stack of the function that identified the issue. This may not be the same function as the source of the error in some cases, such as detecting heap corruption and memory leaks.

The ***Examples of Diagnostic Messages*** table provides three example situations in which diagnostic messages may appear.

Table 3-16: Examples of Diagnostic Messages

Situation	*Warning Text*	*Indication*
Attempting to free a block that is already free	`Heap warning in block 0xFF80647C: free of free blockCall stack: 0xFFA098AC 0xFFA080F6`	The memory block at address `0xFF80647C` has been de-allocated twice.

Table 3-16: Examples of Diagnostic Messages (Continued)

Situation	*Warning Text*	*Indication*
Calling `malloc` with zero size	`Heap warning: allocation of length 0Call stack: 0xFFA09972 0xFFA080F6`	No block address has been provided, as there is no address associated with this issue.
Memory leak	`Heap warning in block 0xFF80647C: unfreed block`	No call stack is displayed here, as it would refer to the call stack of the function in which the leak was detected.

Call Stack

The call stack associated with heap operations is included in the heap debugging output file (and the report generated from the heap debugging output file using the Reporter Tool) or any diagnostic messages produced by the heap debugging library in order to help identify the source of the identified issue. The call stack is stored in a buffer on the system heap, requiring 8 bytes of memory for each potential element in the call stack. By default this call stack is 5 elements deep.

The depth of the call stack can be changed by calling `adi_heap_debug_set_call_stack_depth` at runtime, which will try to re-allocate sufficient space for this buffer. (For more information about `adi_heap_debug_set_call_stack_depth`, refer to the entry in the *C/C++ Library Manual for SHARC Processors*.) You can also keep the original buffer and return false, if it is not possible to change the call stack depth.

The values displayed in the call stack are the PC address of the return from the previous function, starting from the call to the heap function and traversing the stack towards `main` (up to the maximum call stack depth).

For example, when run, the following code:

```
#include <stdlib.h>
#include <heap_debug.h>

void do_free(char *x) {
    free(x);
}

void main(void) {
    adi_heap_debug_enable(_HEAP_STDERR_DIAG);
    do_free(0x0);
}
```

Produces the following warning:

```
Heap warning: free of null pointer Call stack: 0x000400CA 0x000400D7
```

Where the addresses in the call stack, `0x000400CA` and `0x000400D7` refer to the return from the call to `free` in `do_free` and the call to `do_free` from `main`.

Setting the Severity of Error Messages

When `stderr` diagnostics are enabled, the severity of errors can be set based on the type of the error. These severities are described in the ***Heap Debugging Diagnostic Message Severities*** table.

Table 3-17: Heap Debugging Diagnostic Message Severities

Severity	*Description*
Error	The application prints a diagnostic message and terminates.
Ignored	The application does not print any diagnostic message and continues running.
Warning	The application prints a diagnostic message and continues running.

These can be configured at runtime by calling the functions `adi_heap_debug_set_error`, `adi_heap_debug_set_ignore` and `adi_heap_debug_set_warning`, with a parameter that is a bit-field where each bit represents an error type. Macros representing these bits are provided by `heap_debug.h`. Multiple error types can be set to a severity at once by using the bitwise OR operator. (For more information about these functions, refer to the appropriate entries in the *C/C++ Library Manual for SHARC Processors*.)

These priorities have no impact on the report generation; all detected errors are still displayed in the generated report.

If a warning is encountered, but the heap debugging library is unable to use I/O due to being in an interrupt or scheduling being disabled, then the warning is raised to an error and `adi_fatal_error` is called. For this reason, setting the error type `_HEAP_ERROR_ISR` (heap usage within an ISR) to a warning has no effect. Setting `_HEAP_ERROR_ISR` to be ignored behaves as expected. For more information about the `adi_fatal_error` function, refer to the appropriate entry in the *C/C++ Library Manual for SHARC Processors*.

Changing Error Severity Examples

To promote any cases in which the wrong heap is used to de-allocate memory, and any cases of attempting to allocate more memory than the size of the heap so that these cases appear as terminating errors, use the following code:

```
#include <heap_debug.h>

adi_heap_debug_set_error(_HEAP_ERROR_WRONG_HEAP |
                         _HEAP_ERROR_ALLOCATION_TOO_LARGE);
```

To demote any cases of using the wrong function to de-allocate memory, de-allocations of invalid addresses, and heap corruption to a warning, use the following code:

```
#include <heap_debug.h>

adi_heap_debug_set_warning(_HEAP_ERROR_FUNCTION_MISMATCH |
                           _HEAP_ERROR_INVALID_ADDRESS   |
                           _HEAP_ERROR_BLOCK_IS_CORRUPT);
```

To ignore any cases of the wrong heap or wrong function being used to de-allocate memory, use the following code:

```
#include <heap_debug.h>

adi_heap_debug_set_ignore(_HEAP_ERROR_WRONG_HEAP |
                           HEAP_ERROR_FUNCTION_MISMATCH);
```

Default Diagnostic Severities

By default, any potentially suspicious heap behavior documented as acceptable by the run-time libraries or C standard will result in a warning at runtime. Even if this behavior is intentional, it may indicate an error in the usage of the heap, such as attempting to free memory from the wrong heap. Incorrect behavior results in an error at runtime. By default, no issues are ignored.

The default severities of error messages are detailed in the ***Default Heap Debugging Diagnostic Severities*** table.

Table 3-18: Default Heap Debugging Diagnostic Severities

Error Type	*Default Severity*
`_HEAP_ERROR_UNKNOWN`	Error
`_HEAP_ERROR_FAILED`	Warning
`_HEAP_ERROR_ALLOCATION_OF_ZERO`	Warning
`_HEAP_ERROR_NULL_PTR`	Warning
`_HEAP_ERROR_INVALID_ADDRESS`	Error
`_HEAP_ERROR_BLOCK_IS_CORRUPT`	Error
`_HEAP_ERROR_FREE_OF_FREE`	Warning
`_HEAP_ERROR_FUNCTION_MISMATCH`	Error
`_HEAP_ERROR_UNFREED_BLOCK`	Warning
`_HEAP_ERROR_WRONG_HEAP`	Warning
`_HEAP_ERROR_ALLOCATION_TOO_LARGE`	Warning
`_HEAP_ERROR_INVALID_INPUT`	Error
`_HEAP_ERROR_INTERNAL`	Error
`_HEAP_ERROR_IN_ISR`	Error
`_HEAP_ERROR_MISSING_OUTPUT`	Warning
`_HEAP_ERROR_ADDRESSING_MISMATCH`	Warning

For more information on error classes, refer to ***heap_debug.h*** in the *C/C++ Library Manual for SHARC Processors*.

Guard Regions

The heap debugging library uses guard regions of 12 bytes (i.e. 3 words) before and after each block allocated from the heap containing a known bit pattern. These patterns are checked when the free/allocated status of the block is modified or at the end of the application, and if the values do not match, then heap corruption must have occurred, such as overwriting of a buffer or writing to a block that has been de-allocated.

Blocks allocated from the heap have 12 bytes before and after the block filled with the allocated block boundary pattern. Corruption of these before and after guard regions indicates underflow and overflow of the block.

The contents of allocated blocks (other than blocks allocated using `calloc`) are filled with the allocated block contents pattern to help manually identify the use of allocated (but uninitialized) memory.

Free blocks are filled with the free blocks pattern. The 12 byte guard region following the block is also filled with this value, though the 12 byte guard region before the block is not as these 12 bytes are used by the heap for the operation of the free list. Corruption of this memory indicates that memory has been written to after it has been de-allocated.

Reading beyond the scope of the allocated block, of free or uninitialized memory, can be identified by these bit patterns appearing in live data within the application.

The default bit-patterns for the guard regions are shown in the ***Heap Debugging Guard Region Values*** table.

Table 3-19: Heap Debugging Guard Region Values

Guard Region	*Bit Pattern*
Free blocks	`0xBDBDBDBD`
Allocated block boundaries	`0xDDDDDDDD`
Allocated block contents (not `calloc`)	`0xEDEDEDED`
Allocated block contents (`calloc`)	`0x00000000`

These patterns can be changed at runtime by calling

```
bool adi_heap_debug_set_guard_region (unsigned char free-pattern,
unsigned char allocated-pattern, unsigned char content-pattern);
```

where each parameter is a character representing the required bit pattern. Any existing blocks are checked for corruption before the pattern is changed. If there are any corruptions, then `adi_heap_debug_set_guard_region` does not change the guard regions and will return false. If the heap is valid, then the guard regions for all existing allocations are changed along with the guard regions of any future allocations. The patterns written to allocated block contents will not be updated, though any new allocations will be filled with the new bit pattern.

The patterns can also be overridden at build time by defining the appropriate C variable, shown in the ***Heap Debugging Guard Region Variables*** table.

Table 3-20: Heap Debugging Guard Region Variables

Guard Region	*Variable*
Free blocks	`unsigned char adi_heap_guard_free`
Allocated block boundaries	`unsigned char adi_heap_guard_alloc`
Allocated block contents (not `calloc`)	`unsigned char adi_heap_guard_content`

These variables will be updated if `adi_heap_debug_set_guard_region` is called at runtime, so it can be used to identify the current guard region values. For more information about the `adi_heap_debug_set_guard_region` function, refer to the appropriate entry in the *C/C++ Library Manual for SHARC Processors*.

NOTE: The variables described in the ***Heap Debugging Guard Region Variables*** table should not be written to at runtime or false corruption errors may be reported.

The guard regions can be returned to the Analog Devices defaults detailed in the ***Heap Debugging Guard Region Values*** table by calling `adi_heap_debug_reset_guard_region`. (For more information about the `adi_heap_debug_reset_guard_region` function, refer to the appropriate entry in the *C/C++ Library Manual for SHARC Processors*.) As with `adi_heap_debug_set_guard_region`, `adi_heap_debug_reset_guard_region` only changes the guard regions if no corruption has been detected.

Enabling and Disabling Features

There are two ways in which features can be configured within an application: via function calls at runtime or by defining variables at build time. The default configuration is described in Default Behavior.

NOTE: Any allocation or de-allocation made while heap debugging is disabled is not recorded by the heap debugging library. This may result in errors if the memory is then manipulated with heap debugging enabled. For instance, a block allocated with heap debugging disabled and then de-allocated when heap debugging has been enabled will report a free from invalid address error. Conversely, allocation of blocks with heap debugging enabled and consequent manipulation of those blocks with heap debugging disabled may result in an unfreed block error.

The features that can be enabled or disabled, along with the macros provided by `heap_debug.h`, are detailed in the ***Configurable Heap Debugging Features*** table.

Table 3-21: Configurable Heap Debugging Features

Feature	*Macro*
Run-time diagnostics	`_HEAP_STDERR_DIAG`
Generation of `.hpl` file for heap report	`_HEAP_HPL_GEN`
Tracking of heap usage	`_HEAP_TRACK_USAGE`

At Runtime

Features within the heap debugging library can be enabled or disabled at run time by using the functions `adi_heap_debug_enable` or `adi_heap_debug_disable`.

For more information about these functions, refer to the appropriate entries in the *C/C++ Library Manual for SHARC Processors*.

Use a bit-field constructed by combining the required macros specified in in the ***Configurable Heap Debugging Features*** table (see Enabling and Disabling Features), along with the bitwise `or` operator. To enable both run-time diagnostics and `.hpl` file generation, use the following code:

```
adi_heap_debug_enable(_HEAP_STDERR_DIAG | _HEAP_HPL_GEN);
```

Enabling either run-time diagnostics or `.hpl` file generation implicitly enables tracking of heap usage.

At Build Time

The global variables used to configure the heap debugging features can be defined at build time, allowing the default configuration to be modified with no performance overheads. These values can also be read at runtime to identify the current configuration. These variables are detailed in the ***Variables Used To Configure Heap Debugging Features*** table.

NOTE: The variables should not be written to directly at runtime or unexpected behavior may result.

Table 3-22: Variables Used To Configure Heap Debugging Features

Feature	*Variable*
Tracking of heap usage	`bool adi_heap_debug_enabled`
Run-time diagnostics	`bool adi_heap_debug_stderr_diag`
Generation of `.hpl` file for heap report	`bool adi_heap_debug_hpl_gen`

Buffering

The contents of the `.hpl` file used to generate a heap debugging report can be buffered by the heap debugging library. This improves performance and avoids any recorded data being lost when it is not currently safe to write to that file.

The buffer is flushed periodically, or when it is safe to carry out I/O and the buffer has reached a certain threshold.

By default, the heap debugging library does not have a buffer configured. This means that every use of the heap results in the data being written to the output file. As a result, the output file is always up to date and no flushing of the output is required. This does, however, have an impact on execution time. This is due to the overhead of the I/O operations required and means that any data that cannot be written at the time will be lost.

A buffer can be specified at runtime by calling `adi_heap_debug_set_buffer` with a pointer to the memory and the size of the buffer in addressable units. The buffer threshold will be set to half of the size of the buffer. For more information, refer to the `adi_heap_debug_set_buffer` entry in the *C/C++ Library Manual for SHARC Processors*.

A buffer can be configured at built-time by defining the variables described in the ***Variables Used to Configure Heap Debugging Buffer*** table.

Table 3-23: Variables Used to Configure Heap Debugging Buffer

Variable	*Description*
`void *adi_hpl_buf_ptr`	Pointer to the start of the buffer
`int adi_hpl_buf_size`	Size of the buffer in addressable units
`int adi_hpl_buf_threshold`	Threshold at which buffer will be flushed

The macro `_ADI_HEAP_MIN_BUFFER`, provided by `heap_debug.h`, can be used to determine the minimum size required for the heap debugging output buffer to be usable. This macro represents the size required to store 2

entries of the log data along with associated call stacks. The memory requirement for an entry of log data is 56 bytes plus 8 bytes per call stack item, up to the maximum call stack depth. The default maximum call stack depth is 5 and can be modified by using `adi_heap_debug_set_call_stack_depth`.

Refer to the `adi_heap_debug_set_call_stack_depth` entry in the *C/C++ Library Manual for SHARC Processors* for further information.

When heap debugging is not enabled, `_ADI_HEAP_MIN_BUFFER` is defined as `0`.

The number of bytes of data that has been lost due to insufficient buffering is stored in the 32-bit integer variable `adi_hpl_buf_lost_data`, provided by `heap_debug.h`.

Pausing Heap Debugging

Heap debugging can be temporarily disabled at runtime to improve the performance in sections of code where heap usage does not need to be debugged. With debugging disabled, no checks will be carried out and no allocations or de-allocations will be recorded, but performance will be close to the non-debug version of the heap functions.

NOTE: Heap debugging is enabled and disabled globally, so pausing heap debugging will affect the tracking of all heap usage across any running threads until it has been re-enabled.

Any allocations or de-allocations made while heap debugging was paused is not recorded, so any corresponding operations made after heap debugging has been resumed may result in false errors being produced regarding invalid addresses or memory leaks.

Heap debugging can be paused by calling `adi_heap_debug_pause` and can be re-enabled by calling `adi_heap_debug_resume`. For more information about these functions, refer to the appropriate entries in the *C/C++ Library Manual for SHARC Processors*.

Finishing Heap Debugging

If an application does not exit, or uses an OS that does not support `atexit`, the heap debugging library can not clean up or check for corrupt blocks and memory leaks. In these cases the clean-up function `adi_heap_debug_end` should be called at a suitable point within your application. Heap debugging is be disabled upon completion of this function and any further heap usage will be ignored unless heap debugging has been re-enabled by calling `adi_heap_debug_enable`. For more information about these functions, refer to the appropriate entries in the *C/C++ Library Manual for SHARC Processors*.

It is safe to call `adi_heap_debug_end` multiple times within an application.

If an `.hpl` output file has already been written to by the current instance of the application then the output file will be appended to.

NOTE: The `adi_heap_debug_end` function attempts to flush any buffer for the `.hpl` file generation, so should only be called when I/O is safe to use. Calling `adi_heap_debug_end` from within an interrupt or unscheduled region results in `adi_fatal_error` being called.

Verifying Heaps

It is possible to check that one or more heaps are free of corruption at runtime by calling the functions `adi_verify_heap` or `adi_verify_all_heaps`. For more information about these functions, refer to the appropriate entries in the *C/C++ Library Manual for SHARC Processors*. For more information on heap corruption, see Guard Regions.

These functions return true if the heap or heaps are free of corruption, or false if corruption is detected.

Behavior of Heap Debugging Library

The heap debugging library is compatible with the non-debug functionality where possible. This ensures that an application should operate the same with heap debugging enabled as without. However, some minor changes in behavior may be observed. These changes in behavior are detailed in the sections that follow.

Application Size

Due to the additional functionality provided by the heap debugging library, code and data usage for your application will increase when using the heap debugging libraries. Your application may fail to link if insufficient space is available for this library.

Performance

Due to additional validation checks, performance in the heap manipulations is degraded compared to the non-debug version of the functions provided by the C/C++ run-time libraries, especially if generation of the `.hpl` file is enabled. With heap debugging disabled or paused, the performance should be close to the non-debug version of the heap manipulation functions.

Heap debugging can be enabled or disabled at run time, allowing you to ignore selected parts of your applications to minimize the impact of heap performance overheads.

NOTE: Heap operations carried out when heap debugging is disabled are ignored and may result in false errors being reported.

By default, for non-threaded applications, an output file is created. It is used to generate a heap debugging report. The I/O operations required are time consuming and can be disabled to improve performance by using the following code:

```
adi_heap_debug_disable(_HEAP_HPL_GEN);
```

If heap tracing is disabled, then run-time diagnostics should be enabled to identify any heap errors.

Heap Usage

For each allocation on any heap, the heap debugging libraries over allocates the memory by 24 bytes for use as a guard region, as well as approximately 24 bytes of internal data on the system heap. As a result more heap space is used when heap debugging is enabled. You may need to increase the size of your heaps if insufficient space is available.

Stack Usage

The additional function calls used for the heap debugging use the stack of parameters and local variables, so the overall stack usage in your application will increase when using the heap debugging library. This is especially true when writing diagnostics or the `.hpl` file.

realloc

The versions of `realloc` and `heap_realloc` provided by the heap debugging library always de-allocate the original block of memory and allocate a new block of memory of the required size. This is the equivalent of calling `malloc`, `free` then `memcpy`. The non-debug versions of `realloc` and `heap_realloc` try to re-use the existing memory first.

This change in behavior with the heap debugging version is intended to catch cases where a block has been reallocated but pointers haven't been updated to reference the new block. These cases may occur in an application, but this behavior cannot be relied on and may result in unexpected behavior.

As a result, some calls to `realloc` or `heap_realloc` may fail with the heap debugging that are successful without it. This can be avoided by ensuring sufficient heap space is available.

Unfreed File I/O Buffers

For each file stream used, the run-time library allocates 512 bytes of memory from the heap to use as a buffer. For reasons of performance and code size, the run-time libraries do not free this memory upon application exit. The heap debugging library identifies these blocks as belonging to a file buffer so an error about being unfreed will not be reported. The allocation of the I/O buffer memory will be seen in the heap debugging report without a corresponding free.

Memory Used by Operating Systems

Operating systems used in an application may make use of the heaps to store internal data. This data may be reported as an unfreed block by the heap debugging library, as it cannot identify the source of the allocation. Some unfreed block reports are to be expected when using an operating system if it is still running.

Generating a Heap Debugging Report

The default output format for the Heap Debugging report is HTML. To produce the HTML file:

1. Choose ***File*** > ***New*** > ***Code Analysis Report***.

   The ***Analysis Report*** wizard appears.

2. Select ***Heap Debugging*** and click ***Next***.

   The ***Input/output files for code analysis report generation*** page appears.

3. In ***DXE that produced the data file***, enter the name of the application executable file. Alternately, browse to the file.

4. In ***Data file***, enter the name of the heap debugging (`.hpl`) data file. Alternately, browse to the file.

ADDITIONAL INFORMATION: If the application `.ldf` file does not contain an `EXECUTABLE_NAME` directive, the ***Data file*** filed is not populated automatically.

5. In ***Report document's output file name***, enter the name of the new HTML report file. Alternately, browse to the existing file.
6. Click ***Finish***.

To generate a report from the command line, in either pre-formatted HTML or unformatted XML, refer to the Reporter.exe Command Line Report Generation Utility.

Heap Debugging And -char-size-32

For the ADSP-215xx and ADSP-SC5xx processor families which support both byte and word addressing, the heap and heap debugging libraries are available from both addressing modes. The underlying implementation is byte-addressed with a wrapper from word-addressed code. This means that your application will run as expected from either addressing mode or when mixing modes.

The sizes provided in the report are all in bytes, and the guard regions are 12 bytes in length. The addresses reported against the heap activity will reflect the world in which the heap function was called and match the addressed returned to the user application.

The build-time configuration variables are definited from byte addressed code and therefore any source file overriding them must also be built byte addressed (-char-size-8).

NOTE: Any attempt to define the following symbols from word addressed (-char-size-32) code will fail to have any effect: `adi_heap_debug_enabled`, `adi_heap_debug_stderr_diag` , and `adi_heap_debug_hpl_gen`

Attempts to mix addressing modes for heap interaction, such as allocating a block from byte addressed code and deallocating it from word addressed code will report a `_HEAP_ERROR_ADDRESSING_MISMATCH`error.

Stack Overflow Detection

The compiler provides support for detecting stack overflows, which can be particularly troublesome bugs in the limited environment of an embedded system.

This section includes:

- About Stack Overflows gives a description of what a stack overflow is.
- Stack Overflow Detection Facility explains how to use the compiler's support for detecting stack overflows.

About Stack Overflows

This section gives an introduction to stack overflows, and why they are problematic.

This section includes:

- What is Stack Overflow? describes a stack overflow, and why it is different from other bugs.

- Likely Causes of Stack Overflow gives examples of the kind of issues that can lead to stack overflows.
- Difficulties in Diagnosing Stack Overflow shows why compiler support is useful.
- Limitations on the Compiler's Stack Detection Capability notes when compiler support is less useful.
- Fixing a Stack Overflow gives advice on what to do when you have located your stack overflow.

What is Stack Overflow?

A stack overflow is caused by the stack not being large enough for the application. The effects of a stack overflow are undefined; the effects can vary from data corruption to a catastrophic software crash.

The stack overflows when the stack pointer (`I7`) is modified to point past the end of the memory reserved for the stack and the stack is written to using the stack pointer or frame pointer (`I6`).

NOTE: A stack overflow is different from stack corruption caused by a bug in your program code.

When the stack overflows, any writes to the stack using the stack pointer (`I7`) or the frame pointer (`I6`) will begin to corrupt an area of memory which it should not. The results are undefined.

Likely Causes of Stack Overflow

There are many reasons why a stack overflow can occur, for example:

1. A function defines a too-large local array.
2. A function defines a too-large variable-length array. See Variable-Length Array Support.
3. A function uses the `alloca()` function, with an too-large value as its parameter, to allocate space in the stack frame of the caller.
4. The `.ldf` file has insufficient space set aside for the stack.
5. A function calls itself recursively too many times.
6. A function's call tree is too deep.
7. A re-entrant interrupt handler is called too many times before the interrupt is fully serviced.

Note that *too large* or *too many* is only slight more than *not too large* or *not too many*; the application only has to exceed its available stack space by one location for corruption to occur.

Difficulties in Diagnosing Stack Overflow

Without tools for stack overflow detection support, debugging a stack overflow is not often easy and mostly involves setting breakpoints or adding tracing statements at various places in your application. A stack overflow might also not become apparent if you are building your application in a Release configuration, when optimizations are enabled; a stack overflow might not reveal itself until your application is built in a Debug configuration, when optimizations are not enabled.

The timing of interrupts will also mask a stack overflow. If nested interrupts are enabled and the time taken to service the interrupts is insufficient before another interrupt is raised and serviced, then a stack overflow can occur.

Stack Overflow Detection Facility

You can enable stack overflow detection via ***Project*** > ***Properties*** > ***C/C++ Build*** > ***Settings*** > ***Tool Settings*** > ***Compiler*** > ***Run-time Checks*** > ***Generate code to catch a Stack Overflow*** or `-rtcheck-stack`.

The stack is implemented as a circular buffer using `I7`. When stack overflow detection is enabled, the corresponding circular buffer overflow interrupt (`CB7I`) is enabled. If the stack overflows, the interrupt is triggered and transfers control to a function called `_adi_stack_overflowed`. The IDE places a breakpoint on this function automatically, by default. If you hit this breakpoint, examine the `PCSTK` register to determine the interrupt handler return address. The instructions at or just prior to that address that modify `I7` are most probably the cause of the stack overflow.

It is possible that instructions for higher priority interrupts (than `CB7I`) may delay hitting the breakpoint. In this situation it will be necessary to look for a cause of stack overflow in those higher priority handlers or the code immediately before they are raised.

Limitations on the Compiler's Stack Detection Capability

The compiler cannot generate stack overflow detection just for particular files; once stack overflow detection is enabled, it applies to the whole application.

Certain compiler features will cause the compiler to generate calls to support libraries, which will transiently use arbitrarily-deep call-trees, requiring additional stack space. These features are:

- Profiling With Instrumented Code
- Profile-Guided Optimization and Code Coverage
- Heap Debugging

Fixing a Stack Overflow

Once it has been identified that a stack overflow is the cause of your application failure, correcting the problem can be as simple as increasing the amount of memory reserved for your stack.

If, due to hardware memory restrictions, you are unable to increase the amount of memory used for the stack, then conduct a review of your application, examining your use of local arrays, function calling and other program code that leads to a stack overflow.

Reporter.exe Command Line Report Generation Utility

Reporter.exe is the command line utility used to generate reports for Instrumented Profiling, Code Coverage and Heap debugging. The default output format is a pre-formatted HTML report, but the un-formatted XML can be emitted using the `-xml`switch.

The following options are supported:

Switch	Usage
`-dxe <path>`	Path to the executable being profiled. This should not be re-built after generation of the report as the debug info may become out of date.

Switch	Usage
-printer:<printer-name> <data file>[, <data file>]	Specifies the report type to be generated and one or more input data files. For Code Coverage, use -printer:ReporterPGO. For Instrumented Profiling, use -printer:ReporterInstrProf. For Heap Debugging, use -printer:ReporterHeap.
-output <path>	Specifies the path of the generated report.
-xml	Generates an XML report. This is written to the base name of the output file with a .xml extension.
-help	Shows usage and command line information

For example, to produce an XML Code Coverage report, you would invoke the Reporter.exe tool as follows:

```
<path-to-install>/Reporter.exe -dxe Application.dxe -printer:ReporterPGO
Application.pgo -output report.xml -xml
```

Index

Symbols

____ADSP21469_FAMILY__ macro.......................... 2–221
__2116x__ macro.. 2–218
__2126x__ macro.. 2–218
__2136x__ macro.. 2–218
__2137x__ macro.. 2–218
__213xx__ macro.. 2–218
__2146x__ macro.. 2–218
__2147x__ macro.. 2–218,2–222
__2148x__ macro.. 2–218
__214xx__ macro.. 2–218
__ADI_LIBEH__ macro...................................... 2–25
__ADSP21000__ macro...................................... 2–219-2–226
__ADSP21160__ macro...................................... 2–219
__ADSP21160_FAMILY__ macro............................ 2–219
__ADSP21161__ macro...................................... 2–219
__ADSP21161_FAMILY__ macro............................ 2–219
__ADSP2116x__ macro...................................... 2–219
__ADSP211xx__ macro...................................... 2–219
__ADSP21261__ macro...................................... 2–219
__ADSP21262__ macro...................................... 2–219
__ADSP21266__ macro...................................... 2–219
__ADSP21266_FAMILY__ macro............................ 2–219
__ADSP2126x__ macro...................................... 2–219
__ADSP212xx__ macro...................................... 2–220
__ADSP21362__ macro...................................... 2–220
__ADSP21362_FAMILY__ macro............................ 2–220
__ADSP21363__ macro...................................... 2–220
__ADSP21364__ macro...................................... 2–220
__ADSP21365__ macro...................................... 2–220
__ADSP21366__ macro...................................... 2–220
__ADSP21367__ macro...................................... 2–220
__ADSP21367_FAMILY__ macro............................ 2–220,2–221
__ADSP21368__ macro...................................... 2–221
__ADSP21369__ macro...................................... 2–221
__ADSP2136x__ macro...................................... 2–220,2–221
__ADSP21371__ macro...................................... 2–221
__ADSP21371_FAMILY__ macro............................ 2–221
__ADSP21375__ macro...................................... 2–221
__ADSP2137x__ macro...................................... 2–221
__ADSP213xx__ macro...................................... 2–220,2–221
__ADSP21467__ macro...................................... 2–221
__ADSP21469__FAMILY macro............................ 2–221
__ADSP21469__ macro...................................... 2–221
__ADSP21469_FAMILY__ macro............................ 2–222
__ADSP2146x__ macro...................................... 2–221,2–222
__ADSP21477__ macro...................................... 2–222
__ADSP21478__ macro...................................... 2–222
__ADSP21479__ macro...................................... 2–222
__ADSP21479_FAMILY__ macro............................ 2–222,2–223
__ADSP2147x__ macro...................................... 2–222
__ADSP21483__ macro...................................... 2–222
__ADSP21486__ macro...................................... 2–222
__ADSP21487__ macro...................................... 2–222
__ADSP21488__ macro...................................... 2–223
__ADSP21489__ macro...................................... 2–223
__ADSP2148x__ macro...................................... 2–222,2–223
__ADSP214xx__... 2–223
__ADSP214xx__ macro...................................... 2–221-2–223
__ADSP21562__ macro...................................... 2–223
__ADSP21563__ macro...................................... 2–223
__ADSP21565__ macro...................................... 2–223
__ADSP21566__ macro...................................... 2–223
__ADSP21567__ macro...................................... 2–223
__ADSP21569__ macro...................................... 2–224
__ADSP21569_FAMILY__ macro............................ 2–223,2–224
__ADSP2156x__ macro...................................... 2–223,2–224
__ADSP21571__ macro...................................... 2–224
__ADSP21573__ macro...................................... 2–224
__ADSP2157x__ macro...................................... 2–224,2–225
__ADSP21583__ macro...................................... 2–224
__ADSP21584__ macro...................................... 2–224
__ADSP21587__ macro...................................... 2–224
__ADSP2158x__ macro...................................... 2–224-2–226
__ADSP215xx__ macro...................................... 2–223-2–226
__ADSPSC570__ macro...................................... 2–225
__ADSPSC571__ macro...................................... 2–225
__ADSPSC572__ macro...................................... 2–225
__ADSPSC573__ macro...................................... 2–225
__ADSPSC573_FAMILY__ macro............................ 2–224-2–226
__ADSPSC57x__ macro...................................... 2–225,2–226
__ADSPSC582__ macro...................................... 2–225
__ADSPSC583__ macro...................................... 2–225

__ADSPSC583_FAMILY__ macro............................ 2–224
__ADSPSC584__ macro.. 2–226
__ADSPSC587__ macro.. 2–226
__ADSPSC589__ macro.. 2–226
__ADSPSC589_FAMILY__ macro.................. 2–224-2–226
__ADSPSC58x__ macro................................. 2–225,2–226
__ADSPSC5xx__ macro................................. 2–225,2–226
__ADSPSHARC__ macro... 2–226
__alignof__ (type-name) construct............................ 2–209
__ANALOG_EXTENSIONS___ macro.................... 2–226
__attribute__ keyword.. 2–210
__BA_SHARC__ macro................................. 2–223-2–227
__BASE_FILE__ macro... 2–227
__builtin_aligned function.......................... 3–11,3–16,3–37
__builtin_assert() function.. 2–149
__builtin_circindex function.. 3–32
__builtin_circptr function.. 3–32
__BYTE_ADDRESSING__ macro.................... 2–22,2–227
__CCESVERSION__ macro.................................... 2–227
__cplusplus macro... 2–227
__ctor_loop function.. 2–286
__DATE__ macro... 2–227
__DOUBLES_ARE_FLOATS__ macro............ 2–24,2–227
__ECC__ macro.. 2–227
__EDG__ macro... 2–227
__EDG_VERSION__ macro.................................... 2–227
__EXCEPTIONS macro................................. 2–24,2–227
__executable_name symbol.. 3–73
__FILE__ macro... 2–227
__FIXED_POINT_ALLOWED macro...................... 2–227
__FLT64_SHARC__ macro...................................... 2–227
__HETEROGENEOUS_PROCESSOR__ macro...... 2–228
__HOSTNAME__ macro.. 2–50
__IDENT__ macro... 2–228
__lib_setup_processor routine.................................... 2–285
__LINE__ macro.. 2–228
__NORMAL_WORD_CODE__ macro.................... 2–228
__NOSIMD__ macro.. 2–228
__NUM_ARM_CORES__ macro............................ 2–228
__NUM_CORES__ macro....................................... 2–228
__NUM_SHARC_CORES__ macro......................... 2–228
__RTTI macro... 2–228
__SHORT_WORD_CODE__ macro........................ 2–228
__SIGNED_CHARS__ macro.................................. 2–228
__SILICON_REVISION__ macro................... 2–66,2–229
__SIMDSHARC__ macro................... 2–219-2–226,2–229
__STDC__ macro... 2–229
__STDC_VERSION__ macro.................................. 2–229
__SYSTEM__ macro... 2–50
__TIME__ macro... 2–229
__USERNAME__ macro... 2–50
__VERSION__ macro.. 2–229
__VERSIONNUM__ macro..................................... 2–229
__WORKAROUNDS_ENABLED macro 2–66,2–67,2–229
_ADI_COMPILER macro.. 2–226
_ADI_THREADS macro.................................. 2–50,2–226
_HEAP_DEBUG macro... 2–227
_INSTRUMENTED_PROFILING macro................. 2–228
_LANGUAGE_C macro.. 2–228
_LONG_LONG macro... 2–228
_LONG keyword... 2–163
_MISRA_RULES macro.. 2–228
_PGO_HW macro... 2–228
_QUAD keyword... 2–163
_WORD keyword.. 2–163
.doj files... 2–6
.misra extension files... 2–104
.misra files.. 2–54,2–105
.pgi files.. 3–11
.pgo files
  creating data sets... 3–7
  defined.. 2–64
  gathering profile... 3–8
  gathering profile with the -pguide switch................ 2–42
  in PGO process.. 2–42
  -session-id identifier... 2–42
.pgo files: gathering profile.. 2–64
.pgt files... 3–8
.RETAIN_NAME directive... 2–263
.SECTION assembler directive........................ 2–142,2–272
.XML files... 2–31
-@ filename (command file) compiler switch................. 2–19
$OBJS_LIBS_INTERNAL macro.............................. 2–295
operator, used with macro definition....................... 2–208
#define preprocessor command.................................. 2–229
#pragma align alignport... 2–162
#pragma alignment_region... 2–163
#pragma alignment_region_end................................. 2–163
#pragma align num... 3–15
#pragma align num}#pragma align num...................... 2–169
#pragma all_aligned.. 3–37
#pragma alloc.. 3–33

#pragma always_inline....2–21,2–187
#pragma bank_maximum_width.... 2–202
#pragma bank_memory_kind....2–201
#pragma bank_write_cycles.... 2–202
#pragma byte_addressed.... 2–205
#pragma can_instantiate instance....2–184
#pragma code_bank....2–199
#pragma const.... 3–34
#pragma core.... 2–188
#pragma data_bank.... 2–199
#pragma default_addressed.... 2–205
#pragma default_code_bank.... 2–201
#pragma default_data_bank....2–201
#pragma default_section.... 2–192,2–217,2–287
#pragma default_stack_bank....2–201
#pragma diag.... 2–196,3–5
#pragma diag(annotations).... 2–197
#pragma diag(errors)....2–197
#pragma diag(pop)....2–197
#pragma diag(push).... 2–197
#pragma diag(remarks).... 2–197
#pragma diag(warnings)....2–197
#pragma do_not_instantiate instance....2–184
#pragma exceptret....2–173
#pragma file_attr.... 2–195
#pragma function_name....2–188
#pragma FX_CONTRACT....2–185
#pragma FX_ROUNDING_MODE.... 2–97
#pragma inline....2–187
#pragma instantiate....2–287
#pragma instantiate instance....2–184
#pragma interrupt....2–167
#pragma interrupt_complete....2–166
#pragma interrupt_complete_nesting.... 2–167
#pragma interrupt_dispatched_handler.... 2–167
#pragma linkage_name.... 2–185,2–186,2–188
#pragma loop_count(min, max, modulo).... 2–169,3–36
#pragma loop_unroll N}#pragma loop_unroll N....2–169
#pragma misra_func.... 2–174
#pragma no_alias....2–171,3–38
#pragma no_implicit_inclusion.... 2–184
#pragma no_partial_initialization.... 2–165
#pragma no_stack_translation.... 2–205
#pragma no_vectorization....2–169,2–174,3–37
#pragma noreturn....2–174
#pragma once.... 2–185
#pragma optimize_as_cmd_line....2–172
#pragma optimize_for_space....2–172,2–187,3–36
#pragma optimize_for_speed.... 2–172,3–36
#pragma optimize_off....2–172,3–36
#pragma overlay....2–174
#pragma pack (alignopt).... 2–164
#pragma pad (alignopt)....2–165
#pragma param_never_null.... 2–181
#pragma pgo_ignore.... 2–175
#pragma pure....2–175,3–34
#pragma regs_clobbered....2–178,3–35
#pragma regs_clobbered_call.... 2–180
#pragma result_alignment....2–180,3–34
#pragma retain_name.... 2–191
#pragma rtcheck(on)....2–198
#pragma save_restore_40_bits....2–168
#pragma save_restore_simd_40_bits.... 2–168
#pragma section....2–142,2–192,2–217,2–287
#pragma SIMD_for.... 2–168,3–37
#pragma stack_bank.... 2–200
#pragma suppress_null_check....2–182
#pragma system_header.... 2–185
#pragma vector_for....2–172,3–37
#pragma weak_entry....2–195
#pragma word_addressed....2–205
32-bit floating-point arithmetic.... 2–62
40-bit arithmetic....2–212
    implications of using in C/C++ code.... 2–212
    run-time library functions.... 2–212
64-bit floating-point arithmetic.... 2–62
-A (assert) compiler switch....2–19
-absolute-path-dependencies switch.... 2–20

A

accum.... 2–85,2–124,2–270
action qualifier keywords, for use with #pragma diag... 2–196

Symbols

-add-debug-libpaths compiler switch.... 2–20

A

ADSP-21161 processor
    executing code from external SDRAM....2–203
ADSP-2126x/2136x processors
    data placement.... 2–217

data transfer between internal and external memory....... 2–217
ADSP-2146x processors
generating normal word size................................. 2–38
generating short word size.....................................2–50
aggregate assignment support (compiler).................... 2–123
alias, avoiding.. 3–16
alignment inquiry keyword..2–209
alldata section identifier... 2–48
alter macro..2–250
alternate
keywords.. 2–34
registers.. 2–237
alternate heap interface
C++ run-time library support.............................. 2–283
C run-time library functions................................2–282
alternative
operator keywords.. 2–20
tokens, disabling...2–20
tokens, enabling..2–20

Symbols

-alttok (alternative tokens) compiler switch...................2–20
-always-inline compiler switch..................................... 2–21
-anach (enable C++ anachronisms) C++ mode compiler switch...2–56

A

anachronisms
C++ mode compiler switches
-no-anach...2–58
default C++ mode..2–56
disabling in C++ mode... 2–58
-no-anach (disable C++ anachronisms) C++ mode compiler switch..2–58

Symbols

-annotate (enable assembly annotations) compiler switch....... 2–21
-annotate-loop-instr compiler switch................... 2–21,3–61

A

annotation information, instrumental...........................2–21
annotation keyword...2–196
annotations
assembly code...3–56
assembly source code position...............................3–62
disabling...2–21,2–32
enabling...2–48
loop identification... 3–60
modulo scheduling, information provided.............3–65
modulo scheduling, parameters............................ 3–45
vectorization...3–65
anomalies
workarounds...2–66
anomaly_macros_rtl.h.. 2–67
archiver...2–3
arguments and return transfer....................................2–244
argv/argc support...2–287
arithmetic operators for fixed-point types......................2–89
array
storage...2–266
zero length..2–208
arrays
initializer... 2–121
asm
compiler keyword...2–114
keyword...2–209
statement...2–209,3–19
asm_sprt.h system header file......................................2–248

Symbols

-asms-safe-in-simd-for-loops compiler switches.............2–21

A

assembler, for SHARC processors..................................2–3
assembler program... 2–4
assembly
code annotations.. 3–56
support keyword (asm)....................................... 2–254
assembly optimizer
file position..3–62
global information..3–59
loop flattening.. 3–64
loop identification annotation.............................. 3–61
messages and warnings..3–70
modulo scheduling...................................... 3–45,3–65
procedure statistics..3–59
vectorization, annotations.....................................3–65

vectorization, example ... 3–63
assembly output annotations
disabling ... 2–21,2–32
enabling annotations ... 2–21
failure messages ... 3–70
global information ... 3–59
in saved assembly file ... 3–56
loop flattening ... 3–64
loop identification ... 3–60
modulo scheduling ... 3–45,3–65
procedure statistics ... 3–59
selecting ... 3–56
vectorization, defined ... 3–63
warnings ... 3–70
assembly routines
calling from C/C++ program ... 2–248
with exception tables ... 2–263
with parameters example ... 2–259
atexit() library routine ... 2–286
attributes
adding to a file ... 2–294
file ... 2–22,2–26,2–33,2–291
functions, variables and types ... 2–210
names ... 2–291
usage examples ... 2–294
value ... 2–291
attributes, automatically applied ... 2–291

Symbols

-auto-attrs compiler switch ... 2–22

A

autoinit section identifier ... 2–48
automatic
attributes, disabling ... 2–33
attributes, enabling ... 2–22
function inlining ... 2–39
inlining ... 2–64,3–18
inlining, controlled with the - Ov num switch ... 2–40
loop control variables ... 3–29
variables ... 2–135
automatically-applied attributes ... 2–291
automatic attributes
disabling ... 2–32
enabling ... 2–22

B

background registers ... 2–237
bank_memory_kind pragma ... 2–201
bank_optimal_width pragma ... 2–202
bank_read_cycles pragma ... 2–201
bank_write_cycles pragma ... 2–202
bank qualifier ... 2–139
biased round-to-nearest rounding ... 2–97
binary object granularity ... 2–293
bit-fields
unsigned ... 2–51
values ... 2–51
bool, See also Boolean type support keywords ... 2–124
Boolean type support keywords (bool, true, false) ... 2–124
boot loader ... 2–285
bsz section identifier ... 2–48

Symbols

-build-lib (build library) compiler switch ... 2–22

B

build tools ... 2–26
built-in functions
defined ... 2–143
exclusive transactions ... 2–159
expected_false ... 2–148
expected_true ... 2–148
funcsize ... 2–161
in code optimization ... 3–30
iop_read ... 2–160
iop_write ... 2–160
NOP ... 2–160
system support ... 3–30

C

C
tokens in ... 2–20
C/C++
callable subroutines in SIMD mode ... 2–253
code optimization ... 3–1
data types ... 2–59
preprocessor features ... 2–217
run-time model ... 2–231
switch statements ... 2–48

C/C++ compiler, overview.......... 2–3
C/C++ language extensions
bool keyword.......... 2–114
Compound literal expressions.......... 2–114
dm keyword.......... 2–135
false keyword.......... 2–114
indexed initializers.......... 2–114
long identifiers.......... 2–114
non-constant initializers.......... 2–114
pm keyword.......... 2–135
section keyword.......... 2–114
table describing.......... 2–114
true keyword.......... 2–114
variable length arrays.......... 2–114
C/C++ mode selection switches.......... 2–18
-c89.......... 2–18

Symbols

-C (comments) compiler switch.......... 2–22
-c (compile only) compiler switch.......... 2–22

C

C++
alternative tokens in.......... 2–20
class constructor functions.......... 2–48
class instance function.......... 2–246
compiler switches.......... 2–56
constructors and destructors.......... 2–286
exceptions.......... 2–204
member functions in assembly language.......... 2–252
template inclusion control pragma.......... 2–184
templates.......... 2–287
virtual lookup tables.......... 2–48
c++11 compiler switch.......... 2–18
C++ anachronisms
disabling.......... 2–58
enabling.......... 2–56
C++ mode.......... 2–117
C++ mode compiler switches
-anach (enable C++ anachronisms) C++ mode compiler switches.......... 2–56
-check-init-order.......... 2–57,2–286
-eh (enable exception handling).......... 2–24
-full-dependency-inclusion.......... 2–57
-ignore-std.......... 2–57
-no-eh (disable exception handling).......... 2–34
-no-implicit-inclusion.......... 2–58
-no-rtti (disable run-time type identification).......... 2–58
-no-std-templates.......... 2–58
-rtti (enable run-time type identification).......... 2–58
-std-templates.......... 2–58
C++ run-time, alternate heap interface support.......... 2–283
C++ STL objects.......... 2–280
c++ style comments.......... 2–123

Symbols

-c89 (ISO/IEC 9899
1990 standard) compiler switch.......... 2–18

C

C89 mode.......... 2–117

Symbols

-c99 (ISO/IEC 9899
1990 standard) compiler switch.......... 2–18

C

C99 mode.......... 2–117
calling
assembly language subroutine.......... 2–248
assembly language subroutines from C/C++ programs.... 2–248
call preserved registers.......... 2–234,2–261
cc21k compiler
defined.......... 2–1
overview.......... 2–3
ccall macro.......... 2–249,2–255
C compiler
overview.......... 2–101
switches.......... 2–54
C data types.......... 2–266

Symbols

-char-size-8|-32 compiler switch.......... 2–22
-check-init-order C++ mode compiler switch.......... 2–57

C

circular buffers
__builtin_circindex function.......... 3–32
__builtin_circptr function.......... 3–32
automatic generation.......... 2–146

built-in functions....2–146
disabling automatic generation of....2–33
enabling for use....2–26
generating....2–146
increment of index....2–147
increment of pointer....2–147
increments for modulus array references....2–147
indexing....2–146
used in DSP-style code....3–31
used with the -force-circbuf compiler switch....3–31
circular pointer references....2–146
cjump instruction....2–239
C language extensions
C++ style comments....2–115
preprocessor generated warnings....2–114
class conversion optimization pragmas....2–181
classes, initializing global instances....2–286
class pointers, converting....2–181
clobbered
register definition....3–39
registers....2–178
C mode compiler switches
-misra....2–54
-misra-linkdir....2–54
-misra-no-cross-module....2–54
-misra-no-runtime....2–55
-misra-strict....2–55
-misra-suppress-advisory....2–55
-misra-testing....2–55
-Wmis_suppress....2–55
-Wmis_warn....2–55
code_bank pragma....2–199
code generation annotations, enabling....2–52
code inlining, controlling....2–186
CODE memory area....2–287
code optimization
built-in functions....3–30
controlling....3–3
enabling....2–39
for maximum performance....3–33
for size....2–39,3–32
for speed....2–39
using function pragmas....3–33
using loop optimization pragmas....3–36
using pragmas for....3–33
using pragmas in....3–33
with PGO....3–5
code section identifier....2–48
command-line
interface....2–5
comma-separated section qualifiers....2–194
common compiler switches
-no-rtcheck (disable runtime checking)....2–35
-no-rtcheck-arr-bnd (disable runtime checking of array boundaries)....2–35
-no-rtcheck-div-zero (disable runtime checking for division by zero)....2–36
-no-rtcheck-heap (disable runtime checking of heap operations)....2–36
-no-rtcheck-null-ptr (disable runtime checking for NULL pointers)....2–36
-no-rtcheck-shift-check (disable runtime checking of shift values)....2–36
-no-rtcheck-stack (disable runtime checking for stack overflow)....2–36
-no-rtcheck-unassigned (disable runtime checking for unassigned variables)....2–37
-rtcheck (runtime checking)....2–45
-rtcheck-arr-bnd (runtime checking of array boundaries)....2–45
-rtcheck-div-zero (runtime checking for division by zero)....2–46
-rtcheck-heap (runtime checking heap operations)..2–46
-rtcheck-null-ptr (runtime checking for NULL pointers)....2–46
-rtcheck-shift-check (runtime checking of shift values)....2–47
-rtcheck-stack (runtime checking for stack overflow)....2–47
-rtcheck-unassigned (runtime checking for unassigned variables)....2–47

Symbols

-compatible-pm-dm compiler switch....2–22

C

compilation time, indicating with the -no-progress-rep-timeout compiler switch....2–35
compiler
building for a specific hardware revision....2–49,2–65
built-in functions....2–143
C/C++ extensions....2–112,2–114

code generator workarounds.................................. 2–66
code optimization...2–62,3–1
command-line interface, overview........................... 2–5
command-line switch summaries...........................2–18
diagnostic messages..2–196
diagnostics..3–3
disabling GNU compatibility mode.......................2–35
disabling hardware anomaly workarounds..............2–38
enabling GNU compatibility mode....................... 2–31
enabling hardware anomaly workarounds...... 2–53,2–66
keywords, not recognized......................................2–34
optimizer..3–3
overview... 2–3
prelinker...2–64
producing processor-specified code.........................2–43
progress feedback...2–44
selecting specified compilation tool........................2–41
starting a new optimization pass............................ 2–44
stopping after compilation.....................................2–47
undefining macros...2–51
compiler C/C++ extensions..2–114
compiler common switches
-fx-contract (performance and accuracy)................2–27
-fx-rounding-mode-biased................................... 2–27
-fx-rounding-mode-truncation...............................2–27
-fx-rounding-mode-unbiased................................ 2–27
-no-fx-contract.. 2–34
-workaround workaround_id................................2–66
compiler driver.. 2–4,2–66
compiler performance built-in functions
usage example...3–21
compiler proper... 2–4
compiler switches
-component...2–23

Symbols

-component compiler switch...2–23

C

compound literals.. 2–123
compound macros... 2–229
compound statement.. 2–229
conditional code
avoiding in loops.. 3–27
improving...3–20
conditional expressions, with missing operands............2–208
constants
accessed as read-write data.................................... 2–23
initializing statically..3–14
constants, pointer types...2–23
constdata section identifier...2–48

Symbols

-const-read-write compiler switch................................. 2–23

C

constructors, C++ classes.................................2–286,2–287
constructors and destructors
for global class instances...................................... 2–286
memory placement...2–287
start routine...2–286

Symbols

-const-string compiler switch....................................... 2–23

C

content attributes, to map binary objects..................... 2–292
continuation characters..2–35
controlling code inlining...2–186
conversion, fixed-point types.. 2–87
count_ticks function...2–200
CrossCore Embedded Studio
debugger..2–27
graphical user interface (GUI)................................. 2–3
running compiler from command line......................2–3
cross-reference listing information................................2–53
ctdm memory section.. 2–286
custom allocator...2–280

Symbols

-D (define macro) compiler switch.......................2–23,2–51

D

data
alignment pragmas... 2–162
dual-word-aligned...3–15
fetching with 32-bit loads.....................................3–15
memory storage..2–274
storage formats.. 2–266
transfer between internal and external memory.....2–217

word alignment.. 3–15
data_bank pragma.. 2–199
DATA memory area...2–287
data placement
compiler-controlled... 2–48
controlled by the -section id compiler switch........2–143
link-time checking of...2–217
data section identifier..2–48
data storage
formats.. 2–266
initialization.. 2–275
data types
bit sizes..2–59
double.. 2–62
fixed-point..2–84
float..2–61,2–62
formats... 2–266
long double.. 2–62
scalar.. 3–12
sizes.. 2–266
debugging, source-level.. 2–27
debugging information
debug optimization level..2–64
generating..2–27
lightweight.. 2–28
removing.. 2–47
with the -g switch.. 2–27
declarations, mixed with code.................................... 2–122
dedicated registers...2–233
default
heap...2–278
LDF placement... 2–293
names, controlling... 2–143
preprocessor macros, disabling...............................2–33
sections..2–192
default_section pragma... 2–143
definition, unique identifier to....................................2–189
delayed branches, disabled.. 2–33
delete operator
with multiple heaps... 2–283
dependent name processing
disabling..2–58
enabling...2–58
destructors
C++ classes... 2–286
destructors, C++ classes..2–287
diagnostic messages
modifying behavior...2–197
restoring behavior... 2–197
saving behavior... 2–197
severity of... 2–196
diagnostic pragmas
misra_rules_all..2–196
diagnostics
control pragma.. 2–196
described.. 3–3
remarks...3–4
warnings...3–4
diagnostic warnings, enabling....................................... 2–53
Dinkum EC++ Library... 2–3
directive, EXECUTABLE_NAME................................3–73
dm, dual memory support keywords (pm, dm)............2–114
dm, See dual memory support keywords (pm, dm)......2–135
DMAONLY qualifier..2–217
DM qualifier..2–194
double
32-bit data type... 2–22,2–24
64-bit data type... 2–22,2–24
64-bit data type native arithmetic............................. 2–9
data type... 2–266-2–268
data type formats... 2–22,2–24
DOUBLE32 qualifier... 2–194
DOUBLE64 qualifier... 2–194
DOUBLEANY qualifier... 2–194

Symbols

-double-size-32 compiler switch...............2–24,2–266,2–268
-double-size-64 compiler switch............. 2–24,2–266-2–268
-double-size-any compiler switch............ 2–24,2–266,2–268
-dry (verbose dry run) compiler switch.......................... 2–24
-dryrun (terse dry run) compiler switch......................... 2–24

D

dual-memory support keywords (pm dm)....................2–135
dual-word-aligned addresses..3–15
dual-word boundary... 3–16
dynamic_cast run-time type identification.....................2–58

Symbols

-E (stop after preprocessing) compiler switch................. 2–24

E

easm21k assembler....2–3,2–4

Symbols

-ED (run after preprocessing to file) compiler switch.....2–24
-EE (run after preprocessing) compiler switch....2–24
-eh (enable exception handling) C++ mode compiler switch.....2–24

E

elfar archive library....2–3,2–4
elfloader utility....2–275,2–285
emulated arithmetic, avoiding....3–14
entry macro....2–239,2–249
enumeration types....2–25

Symbols

-enum-is-int compiler switch....2–25

E

environment variables
- ADI_DSP....2–59
- CC21K_IGNORE_ENV....2–59
- CC21K_OPTIONS....2–59
- PATH....2–59
- TEMP....2–59
- TMP....2–59

errata workarounds....2–65
error keyword....2–196
error messages
- control pragma....2–196
- overriding....2–52

escape character....2–209
examples, fixed-point dot product....2–86
exception handler
- disabling....2–34
- enabling....2–24

exceptions tables....2–262
exceptret pragma....2–173
exclusive transactions....2–159
EXECUTABLE_NAME directive....3–72,3–73
exit() library routine....2–286
exit macro....2–239,2–249
expected_false built-in function....2–148,3–20
expected_true built-in function....2–148,3–20
EXPRS macro....3–21
external memory
- accessing from processor core....2–217
- accessing with inline functions....2–217
- using the dmaonly keyword with....2–194

Symbols

-extra-precision compiler switch....2–25

F

faster operations, disabling....2–37

Symbols

-fast-fp (fast floating point) compiler switch....2–266

F

file
- annotation position....3–62
- attributes....2–291
  - automatically-applied....2–291
- attributes, adding....2–26
- attributes, disabling....2–33
- automatic attributes....2–22
- extensions....2–7
- multiple attributes....2–26
- searching....2–7

Symbols

-file-attr (file attribute) compiler switch....2–26

F

file name
- description....2–19
- reading from....2–19
- to be processed....2–19

files
- .doj....2–6

file-to-device stream....2–63
fixed-point arithmetic
- pragmas....2–185
- semantics....2–87

fixed-point constants....2–86
fixed-point types
- arithmetic operators....2–89
- conversion....2–87

using..........2–84

Symbols

-flags (command line input) compiler switch..........2–26

F

float
data type..........2–266
float data type..........2–267
floating-point
data size..........2–267
data types..........2–61
hexadecimal constants..........2–122
numbers..........2–266
floating-point multiplication and addition
as associative operations..........2–27
not as associative operations..........2–34
floating-pointunder flow, avoiding..........2–26

Symbols

-float-to-int compiler switch..........2–26

F

float to integer conversion..........2–26

Symbols

-force-circbuf (circular buffer) compiler switch..... 2–26,3–31

F

fract..........2–85,2–124,2–270
frame pointer..........2–237,2–238
free list, emptying..........2–283

Symbols

-full-dependency-inclusion C++ mode compiler switch..2–57

F

FuncName attributes..........2–292
funcsize built-in function..........2–161
function
-always-inline switch..........2–21
arguments/return value transfer..........2–244
calling in loop..........3–28
call return address..........2–256
declarations with pointers..........2–137
entry (prologue)..........2–237,2–256
exit (epilogue)..........2–237,2–256
function call
in loops..........3–28
reported statistics for..........3–59
function call return address..........2–256
function inlining..........2–115
and global asm statements..........2–117
and optimization..........2–116
and run-time checking..........2–118
and sections..........2–117
how to use..........3–18
functions
functsize..........2–161
iop_read..........2–160
iop_write..........2–160
NOP..........2–160
obtaining size in bits..........2–161
function side-effect pragmas
for code optimization..........3–33
listed with example..........2–172
FX_CONTRACT pragma..........2–90,2–185
FX_ROUNDING_MODE pragma..........2–186

Symbols

-fx-contract compiler switch..........2–27
-fx-rounding-mode-biased compiler switch..........2–27
-fx-rounding-mode-truncation compiler switch..........2–27
-fx-rounding-mode-unbiased compiler switch..........2–27
-g (generate debug information) compiler switch..........2–27

G

gets macro..........2–249

Symbols

-glite (lightweight debugging) compiler switch..........2–28

G

global
variables..........2–250
global data..........2–274
global information..........3–59
global variable debugging..........2–27
globvar global variable..........3–29

GNU C compiler....2–206
GNU compatibility mode
  disabling....2–35
  enabling....2–31
granularity....2–293
guard....3–36

Symbols

-H (list headers) compiler switch....2–28

H

hardware
  loops, nested....2–45
  pipelining....3–42
hardware revision, building project for....2–49,2–65
header file control pragmas....2–184
heap
  base address....2–280
  default....2–278
  defining....2–279
  defining at runtime....2–280
  emptying free list....2–283
  freeing space for....2–283
  index....2–282,2–283
  interface, alternate....2–282
  interface, with multiple heaps....2–283
  re-initializing....2–283
heap_calloc function....2–282
heap_free function....2–282
heap_malloc function....2–282
heap_realloc function....2–282,3–95
heap_space_unused function....2–283
heap debugging
  default severities of error messages....3–89
  diagnostic message examples....3–86
  disabling....3–93
  finishing....3–93
  improving performance....3–94
  introduction to....3–81
  library default behavior....3–83
  linking with library....3–82,3–83
  macro....3–82
  non-threaded applications....3–83
  setting severity of errors....3–87
heap debugging library
  buffering contents....3–92
  call stack....3–87
  errors detected by....3–84
  guard regions....3–89
  overheads....3–83
  stderr diagnostics....3–86
heap extension routines
  alternate heap interface....2–282
  heap_calloc....2–278
  heap_free....2–278
  heap_malloc....2–278
  heap_realloc....2–278
  listed....2–278
heap functions
  calloc....2–278
  free....2–278
  malloc....2–278
  realloc....2–278
heaps
  non-default....2–280
  verifying....3–94

Symbols

-help (command-line help) compiler switch....2–28

H

hexadecimal floating-point constants....2–122

Symbols

-HH (list headers and compile) compiler switch....2–28

H

hoisting....3–41

I

I/O conversion specifiers....2–96

Symbols

-I (include search directory) compiler switch....2–28,2–38
-i (less includes) compiler switch....2–29
-I (start include directory) compiler switch....2–29

I

IEEE-754 floating-point formats....2–269

Symbols

-ieee-fp compiler switch.......... 2–266

I

IEEE single-/double-precision formats.......... 2–266

Symbols

-ignore-std C++ mode compiler switch.......... 2–57

I

implicit inclusion
  defined.......... 2–184
  disabling.......... 2–58
  enabling.......... 2–57
  of .cpp files.......... 2–57
implicit instantiation method.......... 2–288
include directory list.......... 2–29
include files, searching.......... 2–29
incomplete function prototype.......... 2–53
index, starting value for.......... 2–147
indexed
  array.......... 3–18
  style.......... 3–18
indexed initializers.......... 2–120
induction variables.......... 3–27
initialization
  data.......... 2–275
  data storage.......... 2–275
  memory.......... 2–31
  order.......... 2–57
  section, processing of.......... 2–275
initializer
  memory.......... 2–31
initializers, indexed.......... 2–120
initiation interval
  described.......... 3–45
  kernel.......... 3–45
inline
  asm statements.......... 3–19
  automatic.......... 3–18
  expansion of C/C++ functions.......... 2–39
  file position.......... 3–62
  function.......... 3–18
  function support keyword, inline.......... 2–114
  keyword, avoiding use of.......... 3–33
  keyword, using.......... 3–19
  qualifier.......... 2–187
inline assembly (add) example.......... 2–254
inline keyword.......... 2–187
inline qualifier
  enabling.......... 2–21
  ignoring.......... 2–32
inlining, with #pragma inline.......... 2–187
inner loops
  improving performance of.......... 3–27
  producing optimal code for.......... 3–36
installation location.......... 2–42
instantiation, template functions.......... 2–183
instrumented profiling.......... 3–74
  about report generation.......... 3–74
  plain text report.......... 3–75
  report contents.......... 3–76
  unexpected results from.......... 3–79
integer data types.......... 2–61,2–266
interface support macros
  alter.......... 2–250
  C/C++ and assembly.......... 2–248
  ccall.......... 2–249
  entry.......... 2–249
  exit.......... 2–249
  gets.......... 2–249
  leaf_entry.......... 2–249
  leaf_exit.......... 2–249
  puts.......... 2–249
  reads.......... 2–249
  restore_reg.......... 2–250
  save_reg.......... 2–250
interfacing C/C++ and assembly, See mixed C/C++/assembly
  programming.......... 2–231
intermediate files, saving.......... 2–48
interprocedural analysis (IPA)
  #pragma core used with.......... 2–188
  code optimization with.......... 2–64
  defined.......... 2–64
  enabling.......... 2–64,3–11
  framework.......... 2–188
  generating usage information.......... 2–64
  identifying variables.......... 3–14
  -ipa compiler switch for.......... 2–64,3–11
interprocedural optimizations.......... 2–64
interrupt handler pragmas.......... 2–165

interrupts, writing in C....2–165
intrinsic (built-in) functions....2–143
iop_read built-in function....2–160
iop_write built-in function....2–160

Symbols

-ipa (interprocedural analysis) compiler switch....2–29,2–64,3–11

I

IPA framework, and #pragma core....2–188
IPA solver utility....2–4
iteration interval....3–45

K

keywords
- alternate....2–34
- compiler....2–113,2–114
- extensions, not recognized....2–34
- extensions, recognized....2–113

keywords (compiler)....2–114

Symbols

-L (library search directory) compiler switch....2–30
-l (link library) compiler switch....2–30
-L (search library) compiler switch....2–38

L

language extensions (compiler)....2–114
leaf_entry macro....2–239,2–249
leaf_exit macro....2–239,2–249
leaf assembly routines....2–254
li1151....2–268
librarian utility....2–4
library
- building with elfar....2–22
- file producing with elfar....2–22
- optimization....2–64
- searching for functions and global variables when linking....2–30

lightweight debugging information....2–28

Symbols

-linear compiler switch....2–30

L

line breaks, in string literals....2–209
line debugging....2–27
linkage_name pragma....2–185
linker and mapping requirements....2–139
linker description file (.ldf file)....2–50
linker utility....2–4
linking pragmas....2–188
link library....2–30

Symbols

-list-workarounds (supported errata workarounds) compiler switch....2–30

L

live register....3–40
loop
- annotations....3–65
- avoiding array writes....3–26
- avoiding conditional code in....3–27
- avoiding function calls in....3–28
- avoiding non-unit strides....3–28
- control variables....3–28
- cycle count....3–61
- epilog....3–40
- exit test....3–29
- flattening....3–64
- identification....3–60
- identification annotation....3–61
- inner vs. outer....3–27
- invariant....3–41
- iteration count....3–36
- kernel....3–40
- optimization, concepts....3–41
- optimization, explained....3–39
- optimization, terminology....3–39
- parallel processing....2–172
- prolog....3–40
- register usage....3–62
- resource usage....3–61
- rotation, defined....3–42
- rotation by hand....3–26
- short....3–25
- trip count....3–28,3–63
- unrolling....3–25

vectorization....2–168,3–37,3–44
loop annotation information
disabling....2–32
enabling....2–21
loop-carried dependency
avoiding loop rotation by hand....3–26
loop optimization pragmas....3–36

Symbols

-loop-simd (generate SIMD code in loops) switch.... 2–30

L

L registers.... 2–178
lvalue
generalized....2–208

Symbols

-M (make only) compiler switch.... 2–31

M

macros
__HOSTNAME__.... 2–50
__SYSTEM__....2–50
__USERNAME__.... 2–50
ccall....2–255
compound statements as....2–229
defining....2–23
expanding to a compound statement.... 2–230
mixed C/C++ assembly support....2–248
predefined preprocessor.... 2–217
stack management.... 2–256
variable argument.... 2–118,2–208
writing....2–229
make rules only....2–31

Symbols

-map (generate a memory map) compiler switch....2–31

M

map files, .XML files....2–31
maximum performance....3–33

Symbols

-MD (make and compile) compiler switch....2–31
-mem (enable memory initialization) compiler switch....2–31

M

mem21k initializer....2–31
disabling....2–35
enabling....2–31
not invoking after linking.... 2–35
processing executable file.... 2–285
processing PROGBITS sections....2–285
processing seg_init initialization section....2–275
processing ZERO_INIT sections....2–285
running.... 2–31
meminit (memory initializer)....2–4
memmove (move memory range) function.... 2–50
memory
bank pragmas.... 2–198
initialization.... 2–31
initialization data storage....2–275
initializer.... 2–31
map file.... 2–31
maximum performance....3–20
space assignments.... 2–137
used for placing code in....2–272
memory bank, max transfer width (bits).... 2–202
memory code/data storage.... 2–274,3–20
memory keywords
function arguments and....2–138
function declarations with pointers....2–137
macros and.... 2–138
memory map, generating.... 2–31
minimum code size, compiling for....3–33
misra_rules_all....2–196
misra_types.h header file....2–106
MISRA C
compiler.... 2–101
compliance.... 2–102
rule 1.4 (required).... 2–104
rule 1.5 (required).... 2–104
rule 10.5 (required).... 2–105
rule 12.12 (required).... 2–106
rule 12.4 (required).... 2–105
rule 12.8 (required).... 2–105
rule 13.2 (advisory)....2–106
rule 13.7 (required).... 2–106
rule 16.10 (required).... 2–106
rule 16.2 (required).... 2–106

rule 16.4 (required)... 2–106
rule 17.1 (required)... 2–106
rule 17.2 (required)... 2–107
rule 17.3 (required)... 2–107
rule 17.4 (required)... 2–107
rule 17.6 (required)... 2–107
rule 18.2 (required)... 2–107
rule 19.15 (advisory)..2–107
rule 19.7 (advisory)..2–107
rule 2.4 (advisory)..2–104
rule 20.10 (required)... 2–108
rule 20.11 (required)... 2–108
rule 20.3 (required)... 2–107
rule 20.4 (required)... 2–108
rule 20.7 (required)... 2–108
rule 20.8 (required)... 2–108
rule 20.9 (required)... 2–108
rule 21.1 (required)... 2–108
rule 5.1 (required)... 2–104
rule 5.5 (advisory)..2–104
rule 5.7 (advisory)..2–104
rule 6.3 (advisory)..2–104
rule 6.4 (advisory)..2–104
rule 8.1 (required)... 2–104
rule 8.10 (required)... 2–105
rule 8.5 (required)... 2–104
rule 8.8 (required)... 2–105
rule 9.1 (required)... 2–105
MISRA C compiler...2–105

Symbols

-misra C mode compiler switch.................................. 2–54

M

MISRA C switches...2–54

Symbols

-misra-linkdir C mode compiler switch.........................2–54
-misra-no-cross-module C mode compiler switch.......... 2–54
-misra-no-runtime C mode compiler switch................. 2–55
-misra-strict C mode compiler switch........................... 2–55
-misra-suppress-advisory C mode compiler switch......... 2–55
-misra-testing C mode compiler switch.........................2–55

M

missing operands, in conditional expressions................2–208
mixed C/C++/assembly programming.........................2–231
mixed C/C++ assembly naming conventions................2–250
mixed C/C++ assembly programming.........................2–231
arguments and return... 2–244
conventions.. 2–231
data storage and type sizes...................................2–266
examples..2–253
return address.. 2–256
stack usage..2–237
mixed C/C++ assembly support, macros..................... 2–248

Symbols

-MM (make rules and compile) compiler switch............2–31
-Mo (processor output file) compiler switch..................2–31

M

MODE1 register..2–166,2–167
modulo, variable expansion unroll factor....................... 3–45
modulo scheduling... 3–45,3–65
modulo scheduling information...................................3–65
modulo variable expansion factor.................................3–52
modulus array references..2–147

Symbols

-Mt filename (output make rule) compiler switch..........2–31

M

multicore support.. 2–188

Symbols

-multiline compiler switch.. 2–31

M

multiple
attributes... 2–26
heaps... 2–278
heap support...2–283
lines, spanning..2–31
multi-statement macros... 2–229

N

naming conventions, assembly and C/C ++................ 2–250

naming conventions, C and assembly........................ 2–250
native fixed-point constants.. 2–86
native fixed-point types, fract and accum..................... 2–124
nested hardware loops, restrictions................................ 2–45

Symbols

-never-inline compiler switch... 2–32

N

newline, in string literals...................................... 2–31,2–35
new operator, with multiple heaps.............................. 2–283
NO_INIT qualifier... 2–194

Symbols

-no-aligned-stack (do not align stack) compiler switch... 2–32
-no-alttok (disable tokens) C++ mode compiler switch.. 2–32
-no-anach (disable C++ anachronisms) compiler switch. 2–58
-no-annotate (disable assembly annotations) compiler common switch... 2–32
-no-annotate-loop-instr compiler common switch......... 2–32
-no-assume-vols-are-iops compiler switch..................... 2–33
-no-auto-attrs compiler switch...................................... 2–33
-no-circbuf (no circular buffer) compiler switch............ 2–33
-no-const-strings compiler switch................................. 2–33
-no-db (no delayed branches) compiler switch.............. 2–33
-no-def (disable definitions) compiler switch................. 2–33
-no-eh (disable exception handling) C++ mode compiler switch.. 2–34
-no-extra-keywords (disable short-form keywords) compiler switch.. 2–34
-no-fp-associative compiler switch................................. 2–34
-no-fx-contract compiler switch.................................... 2–34

N

no implicit inclusion, defined..................................... 2–184

Symbols

-no-implicit-inclusion C++ mode compiler switch......... 2–58
-no-linear compiler switch.. 2–35
-no-main-calls-exit compiler switch.............................. 2–35
-no-mem (disable memory initialization) compiler switch...... 2–35
-no-multiline compiler switch....................................... 2–35

N

non-constant initializer support.................................. 2–120
non-default heap.. 2–280
non-IEEE-754 floating point format.......................... 2–266
non-leaf
  assembly routines... 2–254
  routines to make calls.. 2–260
non-temporary files location... 2–42
non-unit strides, avoiding in loops................................ 3–28
NOP built-in function.. 2–160

Symbols

-no-progress-rep-timeout compiler switch...................... 2–35
-normal-word-code compiler switch.............................. 2–38
-no-rtcheck (disable runtime checking)......................... 2–35
-no-rtcheck-arr-bnd (disable runtime checking of array boundaries).. 2–35
-no-rtcheck-div-zero (disable runtime checking for division by zero).. 2–36
-no-rtcheck-heap (disable runtime checking of heap operations).. 2–36
-no-rtcheck-null-ptr (disable runtime checking for NULL pointers)... 2–36
-no-rtcheck-shift-check (disable runtime checking of shift values)... 2–36
-no-rtcheck-stack (disable runtime checking for stack overflow).. 2–36
-no-rtcheck-unassigned (disable runtime checking for unassigned variables).. 2–37
-no-rtti (disable run-time type identification) C++ mode compiler switch.. 2–58
-no-sat-associative compiler switch................................ 2–37
-no-saturation (no faster operations) compiler switch..... 2–37
-no-shift-to-add compiler switch................................... 2–37
-no-simd (disable SIMD mode) compiler switch........... 2–37
-no-std-ass (disable standard assertions) compiler switch 2–37
-no-std-def (disable standard definitions) compiler switch...... 2–38
-no-std-inc (disable standard include search) compiler switch 2–38
-no-std-lib (disable standard library search) compiler switch.. 2–38
-no-std-templates C++ mode compiler switch............... 2–58
-no-threads (disable thread-safe build) compiler switch.. 2–38
-no-workaround (workaround id) compiler switch......... 2–38

N

num variable....2–39

Symbols

-nwc compiler switch....2–38
-O (enable optimization) compiler switch....2–39
-o (output) compiler switch....2–40
-Oa (automatic function inlining) compiler switch....2–39

O

object files....2–6
operation extensions....2–114,2–115
optimization
- code size....2–39,3–32
- compiler....3–3
- configurations (or levels)....2–62
- default....2–64
- disabling....2–39
- enabling....2–39,2–64
- for code size....3–33
- for maximum performance....3–33
- inner loop....3–27
- interprocedural analysis (IPA)....2–64
- library....2–64
- loops....2–168
- pragmas used in....3–33
- reporting progress in....2–44
- sliding scale for....2–39,2–40
- speed....2–39,3–33
- speed versus size....2–39
- switches....2–39,3–38
- with interprocedural analysis (IPA)....2–64

optimization levels
- automatic inlining....2–64
- debug....2–64
- default....2–64
- interprocedural optimizations....2–64
- PGO....2–64
- procedural optimizations....2–64

Symbols

-Os (optimize for size) compiler switch....2–39

O

outer loops....3–27

Symbols

-overlay (program may use overlays) compiler switch....2–40
-overlay-clobbers compiler switch....2–41

O

overlay pragma....2–174
overlays, registers clobbered by overlay manager....2–41
overlays, using in program....2–40

Symbols

-Ov num (optimize for speed versus size) compiler switch.....2–39
-p (generate instrumented profiling) compiler switch....2–41
-P (omit line numbers and compile) compiler switch....2–41

P

passing
- arguments to driver....2–49
- function parameters....2–244

Symbols

-path- (tool location) compiler switch....2–41
-path-install (installation location) compiler switch....2–42
-path-output (non-temporary files location) compiler switch.....2–42
-path-temp (temporary files location) compiler switch....2–42

P

peeled iterations....3–63
per-file optimizations....2–62,2–64
PGO
- code coverage report....3–80
- collecting data....2–64
- data sets....3–11
- interprocedural optimizations....3–11
- session identifier....2–42

PGO merger....2–4

Symbols

-pgo-session session-id compiler switch
- reference page....2–42
- used to separate profiles....3–10

-pguide (profile-guided optimization) compiler switch....2–42

P

placement
- all data....2–48
- C++ virtual lookup table....2–48
- constant data....2–48
- constant data declared with pm keyword....2–48
- data....2–48,2–143
- initialized data declared with pm keyword....2–48
- initialized variable data....2–48
- initializing aggregate autos....2–48
- jump-tables used to implement C/C++ switch statements....2–48
- machine instructions....2–48
- of run-time library functions....2–291
- static C++ class constructor functions....2–48
- string literals....2–48
- zero-initialized variable data....2–48

placement support keyword (section)....2–114,2–142
pm_constdata section identifier....2–48
pm_data section identifier....2–48
pm, See dual memory support keywords (pm,dm)....2–135
PM qualifier....2–194
pointer
- aligned on dual-word boundaries....3–16
- arithmetic action on....2–209
- class support keyword (restrict)....2–114,2–118
- incrementing....3–18
- induction variable....2–169
- resolving aliasing....3–29
- to data that is aligned....3–16

pointer class support keyword....2–118
pointer-induction variables....2–169
pointer registers....2–236
POP STS instruction....2–166,2–167

Symbols

-pplist (preprocessor listing) compiler switch....2–43

P

pragmas
- data alignment....2–162
- data related....2–199,2–201
- exceptret....2–173
- for default section....2–287
- for fixed-point arithmetic....2–185
- for overlay support....2–174
- for SIMD support....2–168
- for turning off optimization....2–172
- function_name....2–188
- function side-effect....2–172
- FX_CONTRACT....2–185
- FX_ROUNDING_MODE....2–186
- header file control....2–184
- inlining....2–187
- instantiate instance....2–184
- interrupt....2–167
- interrupt_complete....2–166
- interrupt_complete_nesting....2–167
- interrupt_dispatched_handler....2–167
- interrupt handler....2–165
- linkage_name....2–185,2–186,2–188
- linking....2–188
- linking control....2–188
- loop_count (min, max, modulo)....2–169
- loop optimization....2–168,3–36
- memory....2–198,2–201
- multiple cores....2–188
- optimizatio....2–197
- optimization....2–172,2–197
- runtime checking pragmas....2–198
- syntax....2–161
- template instantiation....2–183
- used for code optimization....3–33

predefined macros
- __2116x__....2–218
- __2126x__....2–218
- __2136x__....2–218
- __2137x__....2–218
- __213xx__....2–218
- __2146x__....2–218
- __2147x__....2–218
- __2148x__....2–218
- __214xx__....2–218
- __ADSP21000__....2–219
- __ADSP21160__....2–219
- __ADSP21160_FAMILY__....2–219
- __ADSP21161__....2–219
- __ADSP21161_FAMILY__....2–219
- __ADSP2116x__....2–219
- __ADSP211xx__....2–219
- __ADSP21261__....2–219

__ADSP21262__ 2–219
__ADSP21266__ 2–219
__ADSP21266_FAMILY__ 2–219
__ADSP2126x__ 2–219
__ADSP212xx__ 2–220
__ADSP21362__ 2–220
__ADSP21362_FAMILY__ 2–220
__ADSP21363__ 2–220
__ADSP21364__ 2–220
__ADSP21365__ 2–220
__ADSP21366__ 2–220
__ADSP21367__ 2–220
__ADSP21367_FAMILY__ 2–220
__ADSP21368__ 2–221
__ADSP21369__ 2–221
__ADSP2136x__ 2–221
__ADSP21371__ 2–221
__ADSP21371_FAMILY__ 2–221
__ADSP21375__ 2–221
__ADSP2137x__ 2–221
__ADSP213xx__ 2–221
__ADSP21467__ 2–221
__ADSP21469__ 2–221
__ADSP21469_FAMILY__ 2–222
__ADSP2146x__ 2–222
__ADSP21477__ 2–222
__ADSP21478__ 2–222
__ADSP21479__ 2–222
__ADSP21479_FAMILY__ 2–222
__ADSP2147x__ 2–222
__ADSP21483__ 2–222
__ADSP21486__ 2–222
__ADSP21487__ 2–222
__ADSP21488__ 2–223
__ADSP21489__ 2–223
__ADSP2148x__ 2–223
__ADSP21562__ 2–223
__ADSP21563__ 2–223
__ADSP21565__ 2–223
__ADSP21566__ 2–223
__ADSP21567__ 2–223
__ADSP21569__ 2–224
__ADSP21569_FAMILY__ 2–224
__ADSP2156x__ 2–224
__ADSP21571__ 2–224
__ADSP21573__ 2–224
__ADSP2157x__ 2–224
__ADSP21583__ 2–224
__ADSP21584__ 2–224
__ADSP21587__ 2–224
__ADSP2158x__ 2–224
__ADSP215xx__ 2–225
__ADSPSC570__ 2–225
__ADSPSC571__ 2–225
__ADSPSC572__ 2–225
__ADSPSC573__ 2–225
__ADSPSC573_FAMILY__ 2–226
__ADSPSC57x__ 2–226
__ADSPSC582__ 2–225
__ADSPSC583__ 2–225
__ADSPSC584__ 2–226
__ADSPSC587__ 2–226
__ADSPSC589__ 2–226
__ADSPSC589_FAMILY__ 2–226
__ADSPSC58x__ 2–226
__ADSPSC5xx__ 2–226
__ADSPSHARC__ 2–226
__ANALOG_EXTENSIONS__ 2–226
__BA_SHARC__ 2–227
__BASE_FILE__ 2–227
__BYTE_ADDRESSING__ 2–227
__CCESVERSION__ 2–227
__cplusplus 2–227
__DATE__ 2–227
__DOUBLES_ARE_FLOATS__ 2–227
__ECC__ 2–227
__EDG__ 2–227
__EDG_VERSION__ 2–227
__EXCEPTIONS 2–227
__FILE__ 2–227
__FIXED_POINT_ALLOWED 2–227
__FLT64_SHARC__ 2–227
__HETEROGENEOUS_PROCESSOR__ 2–228
__IDENT__ 2–228
__LINE__ 2–228
__NORMAL_WORD_CODE__ 2–228
__NOSIMD__ 2–228
__NUM_ARM_CORES__ 2–228
__NUM_CORES__ 2–228
__NUM_SHARC_CORES__ 2–228
__RTTI 2–228
__SHORT_WORD_CODE__ 2–228

__SIGNED_CHARS__ 2–228
__SILICON_REVISION__ 2–229
__SIMDSHARC__ 2–229
__STDC__ 2–229
__STDC_VERSION__ 2–229
__TIME__ 2–229
__VERSION__ 2–229
__VERSIONNUM__ 2–229
__WORKAROUNDS_ENABLED 2–229
_ADI_COMPILER 2–226
_ADI_THREADS 2–226
_HEAP_DEBUG 2–227
_INSTRUMENTED_PROFILING 2–228
_LANGUAGE_C 2–228
_LONG_LONG 2–228
_MISRA_RULES 2–54,2–228
_PGO_HW 2–228
predefined preprocessor macros 2–217
prefersMem attribute 2–293
prefersMemNum attribute 2–293
prelinker 2–4,2–64,3–11
MISRA C compiler 2–105
prelinker, generating template files 2–289
preprocessor
listing file 2–43
program 2–217
warnings 2–143
procedural optimizations 2–64
procedure statistics 3–59
processor selection 2–43

Symbols

-proc processor (target processor) compiler switch 2–43
-prof-hw compiler switch 2–43

P

profile-guided optimization (PGO)
adding instrumentation 2–42
described briefly 2–64
flushing 3–9
merger utility 2–4
multiple PGO data sets 3–11
multiple source uses 3–10
-Ov num switch 2–40,3–11,3–33
PGO session identifier 2–42
-pgo-session id switch 2–42
profile instrumentation 2–42
run-time behavior 3–5
usage example 3–23
when not used 2–40
when to use 3–5,3–11
with hardware 3–7
with simulator 3–6
profiling, instrumented 3–74
profiling data, flushing 3–78
PROGBITS section 2–285

Symbols

-progress-rep-func compiler switch 2–44
-progress-rep-opt compiler switch 2–44

P

progress reporting 2–44

Symbols

-progress-rep-timeout compiler switch 2–44
-progress-rep-timeout-secs compiler switch 2–44

P

prototype, incomplete 2–53

Symbols

-p switch, instrumented profiling 3–74

P

PUSH STS instruction 2–166,2–167
puts macro 2–249

Q

QUALIFIER keywords, for section pragma 2–193

Symbols

-R- (disable source path) compiler switch 2–44
-R (search for source files) compiler switch 2–44

R

RAM, initializing 2–285
read_extmem function 2–217
reads macro 2–249
realloc heap function 3–95

ref-code characters........ 2–54
register information
  disabling propagation of........ 2–174
register information, disabling propagation of........ 2–40
registers
  alternate........ 2–237
  clobbered by overlay manager........ 2–41
  clobbered register sets table........ 2–178
  dedicated........ 2–233
  live........ 3–40
  pointer........ 2–236
  reserved........ 2–45
  reserving........ 2–236
  return........ 2–178
  scratch........ 2–235
  soft-wired........ 2–178
  stack........ 2–236
  transfer........ 2–245
  unclobbered........ 2–178
  user........ 2–236
  user-reserved........ 2–178
registers for arguments and return example........ 2–259
regs_clobbered string........ 2–178
remark keyword........ 2–196
remarks
  control pragma........ 2–196
  using in diagnostics........ 3–4
Reporter Tool
  invoking........ 3–75,3–95,3–98

Symbols

-reserve (reserve register) compiler switch........ 2–44,2–236

R

restore_reg macro........ 2–250
restore keyword........ 2–196
restrict
  keyword........ 3–29
  operator keyword........ 2–118
  qualifier........ 3–29
restricted pointer........ 3–29

Symbols

-restrict-hardware-loops compiler switch........ 2–45

R

return address transfer........ 2–256
return registers........ 2–178
return value transfer........ 2–244
rframe instruction........ 2–239
rounding........ 2–97
  biased round-to-nearest........ 2–97
  setting mode........ 2–97
  unbiased round-to-nearest........ 2–97

Symbols

-rtcheck (runtime checking)........ 2–45
-rtcheck-arr-bnd (runtime checking of array boundaries)........ 2–45
-rtcheck-div-zero (runtime checking for division by zero)........ 2–46
-rtcheck-heap (runtime checking of heap operations)........ 2–46
-rtcheck-null-ptr (runtime checking for NULL pointers)........ 2–46
-rtcheck-shift-check (runtime checking of shift values)........ 2–47
-rtcheck-stack (runtime checking for stack overflow)........ 2–47
-rtcheck-unassigned (runtime checking for unassigned variables)........ 2–47
-rtti (enable run-time type identification) C++ mode compiler switch........ 2–58

R

run-time
  C/C++ environment, See mixed C/C++ assembly programming........ 2–231
  C header........ 2–285
  disabling type identification........ 2–58
  dynamically allocate/deallocate memory........ 2–275
  enabling type identification........ 2–58
  stack........ 2–236
RUNTIME_INIT qualifier........ 2–194
runtime checking
  enabling........ 2–109
  limitations........ 2–112
  -no-rtcheck-arr-bnd switch........ 2–35
  -no-rtcheck-div-zero switch........ 2–36
  -no-rtcheck-heap switch........ 2–36
  -no-rtcheck-null-ptr switch........ 2–36
  -no-rtcheck-shift-check switch........ 2–36
  -no-rtcheck-stack switch........ 2–36

-no-rtcheck switch 2–35
-no-rtcheck-unassigned switch 2–37
pragmas 2–110,2–198
response upon detection 2–111
-rtcheck-arr-bnd switch 2–45
-rtcheck-div-zero switch 2–46
-rtcheck-heap switch 2–46
-rtcheck-null-ptr switch 2–46
-rtcheck-shift-check switch 2–47
-rtcheck-stack switch 2–47
-rtcheck switch 2–45
supported checks 2–110
runtime checking: pragmas 2–198

Symbols

-S (stop after compilation) compiler switch 2–47
-s (strip debug information) compiler switch 2–47
-sat-associative compiler switch 2–48

S

saturation
disabling 2–37
disabling associativity 2–37
enabling associativity 2–48
save_reg macro 2–250

Symbols

-save-temps (save intermediate files) compiler switch 2–48

S

scheduling, of program instructions 3–40
scratch registers 2–235
scratch registers (dot product) example 2–257
search path
for include files 2–28
for library files 2–30
for library files when linking 2–30
secondary registers 2–237
section
elimination 3–32
keyword 2–114,2–142
placing symbols in 2–192
qualifiers 2–192

Symbols

-section id (data placement) compiler switch 2–48,2–143,2–287

S

section identifiers, compiler-controlled 2–48
section keyword 2–114
section pragmas 2–192
SECTKIND keywords 2–193
SECTSTRING double-quoted string 2–193
seg_dmda_bw memory section 2–273
seg_dmda_nw memory section 2–273
seg_dmda data section 2–274
seg_dmda memory section 2–273
seg_init initialization section 2–275
seg_init memory section 2–273
seg_int_code_sw memory section 2–273
seg_int_code memory section 2–273
seg_pmco code section 2–274
seg_pmco memory section 2–272
seg_pmda_bw memory section 2–273
seg_pmda_nw memory section 2–273
seg_pmda data section 2–274
seg_pmda memory section 2–273
seg_rth memory section 2–273
seg_swco memory section 2–273
segment
keyword 2–114
SHARC processors, data types 2–266
short-form keywords
disabling 2–34

Symbols

-short-word-code compiler switch 2–49
-show (display command line) compiler switch 2–49
-signed-bitfield compiler switch 2–49

S

silicon revision
version setting 2–65
silicon revision, specifying 2–49,2–65
SIMD_for pragma 2–168
SIMD mode
C/C++ callable subroutines 2–253
disabling 2–37

single case range..2–209
sinking process...3–41

Symbols

-si-revision (silicon revision) compiler switch........ 2–49,2–65

S

sizeof() operator.. 2–209
sliding scale, between 0 and 100.................................. 2–39
small applications, producing..3–32
software pipelining...3–42,3–44
source code, checking for syntax errors..........................2–50
source directory
    adding.. 2–44
source path
    disabling...2–44
spill, to the stack...3–40
stack
    frame, defined.. 2–238
    managing in memory..2–237
    managing routines... 2–237
    managing with macros.......................................2–256
    overflow detection... 3–96
    pointer..2–237
    registers.. 2–236
stack_bank pragma... 2–200
stack alignment, disabling..2–32
stack for arguments and return example.......................2–259
stack overflow
    about..3–96
    causes.. 3–97
    detecting..3–98
    detection..3–96
    difficulties..3–97
stage count (SC)... 3–45,3–48
standard
    include search, disabling...................................... 2–38
    library search, disabling....................................... 2–38
    macro definitions, disabling.................................2–38
standard assertions
    disabling...2–37
    enabling...2–19
statement expression.. 2–206
static data..2–274
status register, saving data in...................................... 2–167
STDC STDC FX_FULL_PRECISION pragma......... 2–186

Symbols

-std-templates C++ mode compiler switch..................... 2–58

S

STI memory area..2–287
sti section identifier..2–48
string literals
    marking as const-qualified..................................... 2–23
    no-multiline... 2–35
    not making const-qualified.................................... 2–33
    with line breaks... 2–209
strtofxfx (convert string to fixed-point) function............2–96
struct
    assignment..2–50
    copying...2–50
struct/unions... 2–211

Symbols

-structs-do-not-overlap compiler switch........................ 2–50

S

structures
    initializing.. 2–121
subroutine return address, example............................. 2–255
suppress keyword...2–196

Symbols

-swc compiler switch...2–50

S

switches
    -@filename (command file).................................. 2–19
    -A (assert) compiler switch....................................2–19
    -absolute-path-dependencies.................................2–20
    -add-debug-libpaths..2–20
    -alttok (alternative tokens)....................................2–20
    -always-inline...2–21
    -annotate (enable assembly annotations).................2–21
    -annotate-loop-instr..2–21
    -asms-safe-in-simd-for-loops.................................2–21
    -auto-attrs (automatic attributes)........................... 2–22
    -build-lib (build library).. 2–22
    -C (comments)... 2–22
    -c (compile only)... 2–22

-compatible-pm-dm.. 2–22
-const-read-write...2–23
-const-strings... 2–23
-D (define macro)...2–23
-double-size-32|64.. 2–22,2–24
-double-size-any...2–24
-dry (verbose dry-run)... 2–24
-dryrun (terse dry-run)... 2–24
-E (stop after preprocessing).................................. 2–24
-ED (run after preprocessing to file)....................... 2–24
-EE (run after preprocessing).................................. 2–24
-enum-is-int... 2–25
-extra-precision...2–25
-file-attr name...2–26
-flags (command-line input).................................. 2–26
-float-to-int...2–26
-force-circbuf...2–26
-fp-associative (floating-point associative operation).......
....2–27
-full-version (display versions)................................2–27
-g (generate debug information)............................. 2–27
-glite (lightweight debugging).................................2–28
-H (list headers)..2–28
-help (command-line help).................................... 2–28
-HH (list headers and compile).............................. 2–28
-i (less includes)... 2–29
-I (start include directory).......................................2–29
-I directory (include search directory)..................... 2–28
-include (include file).. 2–29
-ipa (interprocedural analysis)................................2–29
-L (library search directory)................................... 2–30
-l (link library)...2–30
-linear-simd... 2–30
-list-workarounds (supported errata workarounds)..2–30
-loop-simd...2–30
-M (generate make rules only)............................... 2–31
-map filename (generate a memory map)................2–31
-MD (generate make rules and compile).................2–31
-mem (enable memory initialization)......................2–31
-MM (generate make rules and compile)................ 2–31
-Mo (processor output file)....................................2–31
-Mt filename (output make rule)........................... 2–31
-multiline.. 2–31
-never-inline.. 2–32
-no-aligned-stack (disable stack alignment).............2–32
-no-alttok (disable alternative tokens).................... 2–32
-no-annotate (disable alternative tokens).................2–32
-no-annotate-loop-instr... 2–32
-no-assume-vols-are-iops..2–33
-no-circbuf (no circular buffer).............................. 2–33
-no-const-strings..2–33
-no-db (no delayed branches)................................ 2–33
-no-defs (disable defaults)..................................... 2–33
-no-extra-keywords (disable short-form keywords)..2–34
-no-fp-associative...2–34
-no-linear-simd..2–34
-no-loop-simd..2–34
-no-main-calls-exit...2–35
-no-mem (disable memory initialization)................2–35
-no-multiline... 2–35
-no-progress-rep-timeout....................................... 2–35
-normal-word-code..2–38
-no-sat-associative..2–37
-no-saturation (no faster operations).......................2–37
-no-shift-to-add... 2–37
-no-simd (disable SIMD mode)..............................2–37
-no-std-ass (disable standard assertions)................. 2–37
-no-std-def (disable standard macro definitions)..... 2–38
-no-std-inc (disable standard include search)...........2–38
-no-std-lib (disable standard library search)............2–38
-no-threads (disable thread-safe build)................... 2–38
-no-workaround (workaround id)...........................2–38
-nwc.. 2–38
-O (enable optimizations).......................................2–39
-o (output file)...2–40
-Oa (automatic function inlining)......................... 2–39
-Os (optimize for size)... 2–39
-overlay...2–40
-overlay-clobbers..2–41
-Ov num (optimize for speed vs. size).....................2–39
-p (generate instrumented profiling).......................2–41
-P (omit line numbers and compile)....................... 2–41
-path- (tool location).. 2–41
-path-install (installation location)..........................2–42
-path-output (non-temporary files location)........... 2–42
-path-temp (temporary files location)..................... 2–42
-pgo-session session-id.. 2–42
-pguide (profile-guided optimization).....................2–42
-PP (omit line numbers and compile).....................2–41
-pplist (preprocessor listing)...................................2–43
-proc processor.. 2–43
-progress-rep-func..2–44

-progress-rep-opt.. 2–44
-progress-rep-timeout.. 2–44
-progress-rep-timeout-secs.................................... 2–44
-R (add source directory)...................................... 2–44
-R- (disable source path).. 2–44
-reserve (reserve register)....................................... 2–44
-restrict-hardware-loops... 2–45
-S (stop after compilation)..................................... 2–47
-s (strip debug information)................................... 2–47
-sat-associative... 2–48
-save-temps (save intermediate files)....................... 2–48
-section id (data placement)................................... 2–48
-short-word-code.. 2–49
-show (display command line)............................... 2–49
-si-revision version (silicon revision)............. 2–49,2–65
sourcefile (parameter)... 2–19
-structs-do-not-overlap.. 2–50
-swc.. 2–50
-syntax-only (check syntax only)............................ 2–50
-syntax-only (system definitions)........................... 2–50
-T filename (.ldf file)... 2–50
-threads (enable thread-safe build)......................... 2–50
-time (tell time)... 2–51
-U (undefine macro).. 2–51
-unsigned-bitfield (make plain bit-fields unsigned). 2–51
-v (version and verbose)... 2–52
-verbose... 2–52
-version (display version)....................................... 2–52
-w (disable all warnings).. 2–53
-W{...} number (override error message)................ 2–52
-Wannotations (enable code generation annotations)...... 2–52
-warn-protos (warn if incomplete prototype).......... 2–53
-Werror-limit (maximum compiler errors).............. 2–52
-Werror-warnings (treat warnings as errors)............ 2–52
-workaround workaround_id................................ 2–53
-Wremarks (enable diagnostic warnings)................ 2–53
-Wterse (enable terse warnings).............................. 2–53
-xref (cross-reference list)....................................... 2–53
switchescompiler switch
-signed-bitfield.. 2–49
switch section identifier... 2–48
symbols, placing in sections....................................... 2–192

Symbols

-syntax-only (check syntax only) compiler switch...........2–50
-sysdef (system definitions) compiler switch..................2–50

S

sysreg_bit_clr_nop function..2–146
sysreg_bit_clr function..2–146
sysreg_bit_set_nop function...................................... 2–146
sysreg_bit_set function..2–146
sysreg_bit_tgl_nop function..2–146
sysreg_bit_tgl function..2–146
sysreg_bit_tst_all function.. 2–146
sysreg_bit_tst function..2–146
sysreg_read function.. 2–146
sysreg_write_nop function... 2–146
sysreg_write function... 2–146
sysreg.h header file... 2–146,3–30
system macros, defined.. 2–50
system registers
accessing.. 2–146
handling... 3–30
list of.. 2–146

T

target processor, specifying...2–43
template
class.. 2–287
control pragma.. 2–184
function...2–287
instantiation pragmas...2–183
support in C++.. 2–287
un-instantiated.. 2–289
template instantiation... 2–4,2–288
temporary files location..2–42

Symbols

-T filename (linker description file) compiler switch...... 2–50
-threads (enable thread-safe build) compiler switch........ 2–50

T

thread-safe build, disabling.. 2–38

Symbols

-time (tell time) compiler switch.................................. 2–51

T

transfer registers........ 2–245
transferring
  function arguments and return value........ 2–244
  function parameters to assembly routines........ 2–244
trip
  count........ 3–45,3–53
  loop count........ 3–63
  maximum........ 3–45
  minimum........ 3–45
  modulo........ 3–45
truncation........ 2–97
type cast........ 2–209
type sizes, data........ 2–266

Symbols

-U (undefine macro) compiler switch........ 2–23,2–51

U

unbiased round-to-nearest rounding........ 2–97
unclobbered registers........ 2–178
unnamed struct/union fields........ 2–211

Symbols

-unsigned-bitfield (make plain bit-fields unsigned) compiler switch........ 2–51

U

user identifier........ 2–279
user registers........ 2–236

V

va_arg (get next argument in variable list) function........ 2–245
va_start (set variable list pointer) function........ 2–245
variable
  argument macros........ 2–118,2–208
  length array........ 2–208
  name length........ 2–143
  statically initialized........ 3–14
variable argument list
  details of argument passing........ 2–245
variable expansion and MVE unroll........ 3–49
vectorization
  annotations........ 3–65
  avoiding........ 3–37
  defined........ 3–63
  factor........ 3–63
  loop........ 3–37,3–44
  transformation........ 3–37

Symbols

-verbose (display command line) compiler switch........ 2–52
-version (display version) compiler switch........ 2–52

V

version information, displaying........ 2–27
virtual function lookup tables........ 2–48,2–143
VISA (Variable Instruction Set Architecture)........ 2–8
void functions examples........ 2–258
volatile
  about........ 3–12
  declarations........ 3–3
  register set........ 2–178
vtable section identifier........ 2–48
vtbl section identifier........ 2–48,2–143

Symbols

-w (disable all warnings) compiler switch........ 2–53,3–4
-W{...} number (override error message) compiler switch........ 2–52,3–4
-Wannotations (enable diagnostic warnings) compiler switch........ 2–52
-warn-component compiler switch
  compiler switches -warn-component........ 2–53

W

warning keyword........ 2–196
warning messages
  #warning directive........ 2–143
  control pragma........ 2–196
  described........ 3–4
  diagnostic........ 3–4
  disabling........ 3–4
  disabling all........ 2–53
  errors........ 2–52

Symbols

-Warn-protos (warn if incomplete prototype) compiler switch........ 2–53

-Werror-limit (maximum compiler errors) compiler switch.... 2–52
-Werror-warnings compiler switch.... 2–52
-Wmis_suppress C mode compiler switch.... 2–55
-Wmis_warn C mode compiler switch.... 2–55

W

workarounds
- enabling.... 2–66
- errata.... 2–65
- interaction between -si-revision, -workaround, and -no-workaround.... 2–67
- list of valid workarounds.... 2–66
- use of the -workaround switch.... 2–66
- valid workarounds list.... 2–66
- -workaround switch.... 2–66

Symbols

-workaround workaround_id compiler switch.......2–53,2–66
-Wremarks (enable diagnostic remarks) compiler switch 2–53
-Wremarks (enable diagnostic warnings) compiler switch....... 2–53,3–4

W

write_extmem function.... 2–217
writes, array element.... 3–26

Symbols

-Wterse (enable terse warnings) compiler switch.... 2–53
-xref (cross-reference list) compiler switch.... 2–53

Z

ZERO_INIT qualifier.... 2–194,2–292
ZERO_INIT section.... 2–285
zero length arrays.... 2–208