

ARM[®] Compiler toolchain

Version 5.02

Compiler Reference



ARM Compiler toolchain

Compiler Reference

Copyright © 2010-2012 ARM. All rights reserved.

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
28 May 2010	A	Non-Confidential	ARM Compiler toolchain v4.1 Release
30 September 2010	B	Non-Confidential	Update 1 for ARM Compiler toolchain v4.1
28 January 2011	C	Non-Confidential	Update 2 for ARM Compiler toolchain v4.1 Patch 3
30 April 2011	D	Non-Confidential	ARM Compiler toolchain v5.0 Release
29 July 2011	E	Non-Confidential	Update 1 for ARM Compiler toolchain v5.0
30 September 2011	F	Non-Confidential	ARM Compiler toolchain v5.01 Release
29 February 2012	G	Non-Confidential	Document update 1 for ARM Compiler toolchain v5.01 Release
27 July 2012	H	Non-Confidential	ARM Compiler toolchain v5.02 Release

Proprietary Notice

Words and logos marked with [™] or [®] are registered trademarks or trademarks of ARM in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Some material in this document is based on IEEE 754 - 1985 IEEE Standard for Binary Floating-Point Arithmetic. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner.

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

<http://www.arm.com>

Contents

ARM Compiler toolchain Compiler Reference

Chapter 1	Conventions and Feedback	
Chapter 2	Introduction	
	2.1 About the ARM compiler	2-2
	2.2 Source language modes of the compiler	2-3
	2.3 ISO C90	2-4
	2.4 ISO C99	2-5
	2.5 ISO C++	2-6
	2.6 Language extensions and language compliance	2-7
	2.7 Language extensions	2-8
	2.8 Language compliance	2-9
	2.9 The C and C++ libraries	2-10
Chapter 3	Compiler Command-line Options	
	3.1 -Aopt	3-6
	3.2 --allow_fpreg_for_nonfpdata, --no_allow_fpreg_for_nonfpdata	3-7
	3.3 --allow_null_this, --no_allow_null_this	3-8
	3.4 --alternative_tokens, --no_alternative_tokens	3-9
	3.5 --anachronisms, --no_anachronisms	3-10
	3.6 --apcs=qualifer...qualifier	3-11
	3.7 --arm	3-15
	3.8 --arm_linux	3-16
	3.9 --arm_linux_config_file=path	3-18
	3.10 --arm_linux_configure	3-19
	3.11 --arm_linux_paths	3-21
	3.12 --arm_only	3-23
	3.13 --asm	3-24
	3.14 --asm_dir=directory_name	3-25
	3.15 --autoinline, --no_autoinline	3-26

3.16	--bigend	3-27
3.17	--bitband	3-28
3.18	--brief_diagnostics, --no_brief_diagnostics	3-29
3.19	--bss_threshold=num	3-30
3.20	-c	3-31
3.21	-C	3-32
3.22	--c90	3-33
3.23	--c99	3-34
3.24	--code_gen, --no_code_gen	3-35
3.25	--compatible=name	3-36
3.26	--compile_all_input, --no_compile_all_input	3-37
3.27	--conditionalize, --no_conditionalize	3-38
3.28	--configure_cpp_headers=path	3-39
3.29	--configure_extra_includes=paths	3-40
3.30	--configure_extra_libraries=paths	3-41
3.31	--configure_gas=path	3-42
3.32	--configure_gcc=path	3-43
3.33	--configure_gcc_version=version	3-44
3.34	--configure_gld=path	3-45
3.35	--configure_sysroot=path	3-46
3.36	--cpp	3-47
3.37	--cpu=list	3-48
3.38	--cpu=name	3-49
3.39	--create_pch=filename	3-53
3.40	-Dname[(parm-list)][=def]	3-54
3.41	--data_reorder, --no_data_reorder	3-55
3.42	--debug, --no_debug	3-56
3.43	--debug_macros, --no_debug_macros	3-57
3.44	--default_definition_visibility=visibility	3-58
3.45	--default_extension=ext	3-59
3.46	--dep_name, --no_dep_name	3-60
3.47	--depend=filename	3-61
3.48	--depend_dir=directory_name	3-62
3.49	--depend_format=string	3-63
3.50	--depend_single_line, --no_depend_single_line	3-65
3.51	--depend_system_headers, --no_depend_system_headers	3-66
3.52	--depend_target=target	3-67
3.53	--device=list	3-68
3.54	--device=name	3-69
3.55	--diag_error=tag[,tag,...]	3-70
3.56	--diag_remark=tag[,tag,...]	3-71
3.57	--diag_style={arm ide gnu}	3-72
3.58	--diag_suppress=tag[,tag,...]	3-73
3.59	--diag_suppress=optimizations	3-74
3.60	--diag_warning=tag[,tag,...]	3-75
3.61	--diag_warning=optimizations	3-76
3.62	--dllexport_all, --no_dllexport_all	3-77
3.63	--dllimport_runtime, --no_dllimport_runtime	3-78
3.64	--dollar, --no_dollar	3-79
3.65	--dwarf2	3-80
3.66	--dwarf3	3-81
3.67	-E	3-82
3.68	--echo	3-83
3.69	--emit_frame_directives, --no_emit_frame_directives	3-84
3.70	--enum_is_int	3-85
3.71	--errors=filename	3-86
3.72	--exceptions, --no_exceptions	3-87
3.73	--exceptions_unwind, --no_exceptions_unwind	3-88
3.74	--execstack, --no_execstack	3-89
3.75	--export_all_vtbl, --no_export_all_vtbl	3-90

3.76	--export_defs_implicitly, --no_export_defs_implicitly	3-91
3.77	--extended_initializers, --no_extended_initializers	3-92
3.78	--feedback=filename	3-93
3.79	--force_new_nothrow, --no_force_new_nothrow	3-94
3.80	--forceinline	3-95
3.81	--fp16_format=format	3-96
3.82	--fpmode=model	3-97
3.83	--fpu=list	3-99
3.84	--fpu=name	3-100
3.85	--friend_injection, --no_friend_injection	3-104
3.86	-g	3-105
3.87	--global_reg=reg_name[,reg_name,...]	3-106
3.88	--gnu	3-107
3.89	--gnu_defaults	3-108
3.90	--gnu_instrument, --no_gnu_instrument	3-109
3.91	--gnu_version=version	3-110
3.92	--guiding_decls, --no_guiding_decls	3-111
3.93	--help	3-112
3.94	--hide_all, --no_hide_all	3-113
3.95	-ldir[,dir,...]	3-114
3.96	--ignore_missing_headers	3-115
3.97	--implicit_include, --no_implicit_include	3-116
3.98	--implicit_include_searches, --no_implicit_include_searches	3-117
3.99	--implicit_key_function, --no_implicit_key_function	3-118
3.100	--implicit_typename, --no_implicit_typename	3-119
3.101	--import_all_vtbl	3-120
3.102	--info=totals	3-121
3.103	--inline, --no_inline	3-122
3.104	--interface_enums_are_32_bit	3-123
3.105	--interleave	3-124
3.106	-Jdir[,dir,...]	3-125
3.107	--kandr_include	3-126
3.108	-Lopt	3-127
3.109	--library_interface=lib	3-128
3.110	--library_type=lib	3-130
3.111	--licretry	3-131
3.112	--link_all_input, --no_link_all_input	3-132
3.113	--list	3-133
3.114	--list_dir=directory_name	3-135
3.115	--list_macros	3-136
3.116	--littleend	3-137
3.117	--locale=lang_country	3-138
3.118	--long_long	3-139
3.119	--loose_implicit_cast	3-140
3.120	--lower_ropi, --no_lower_ropi	3-141
3.121	--lower_rwpi, --no_lower_rwpi	3-142
3.122	--ltcg	3-143
3.123	-M	3-144
3.124	--md	3-145
3.125	--message_locale=lang_country[.codepage]	3-146
3.126	--min_array_alignment=opt	3-147
3.127	--mm	3-148
3.128	--multibyte_chars, --no_multibyte_chars	3-149
3.129	--multifile, --no_multifile	3-150
3.130	--multiply_latency=cycles	3-151
3.131	--narrow_volatile_bitfields	3-152
3.132	--nonstd_qualifier_deduction, --no_nonstd_qualifier_deduction	3-153
3.133	-o filename	3-154
3.134	-Onum	3-156
3.135	--old_specializations, --no_old_specializations	3-158

3.136	--old_style_preprocessing	3-159
3.137	-Ospace	3-160
3.138	-Otime	3-161
3.139	--output_dir=directory_name	3-162
3.140	-P	3-163
3.141	--parse_templates, --no_parse_templates	3-164
3.142	--pch	3-165
3.143	--pch_dir=dir	3-166
3.144	--pch_messages, --no_pch_messages	3-167
3.145	--pch_verbose, --no_pch_verbose	3-168
3.146	--pending_instantiations=n	3-169
3.147	--phony_targets	3-170
3.148	--pointer_alignment=num	3-171
3.149	--preinclude=filename	3-172
3.150	--preprocess_assembly	3-173
3.151	--preprocessed	3-174
3.152	--project=filename, --no_project	3-175
3.153	--protect_stack, --no_protect_stack	3-176
3.154	--reassociate_saturation, --no_reassociate_saturation	3-177
3.155	--reduce_paths, --no_reduce_paths	3-178
3.156	--reinitialize_workdir	3-179
3.157	--relaxed_ref_def, --no_relaxed_ref_def	3-180
3.158	--remarks	3-181
3.159	--remove_unneeded_entities, --no_remove_unneeded_entities	3-182
3.160	--restrict, --no_restrict	3-183
3.161	--retain=option	3-184
3.162	--rtti, --no_rtti	3-185
3.163	--rtti_data, --no_rtti_data	3-186
3.164	-S	3-187
3.165	--shared	3-188
3.166	--show_cmdline	3-189
3.167	--signed_bitfields, --unsigned_bitfields	3-190
3.168	--signed_chars, --unsigned_chars	3-191
3.169	--split_ldm	3-192
3.170	--split_sections	3-193
3.171	--strict, --no_strict	3-194
3.172	--strict_warnings	3-195
3.173	--sys_include	3-196
3.174	--thumb	3-197
3.175	--translate_g++	3-198
3.176	--translate_gcc	3-200
3.177	--translate_gld	3-202
3.178	--trigraphs, --no_trigraphs	3-204
3.179	--type_traits_helpers, --no_type_traits_helpers	3-205
3.180	-Uname	3-206
3.181	--unaligned_access, --no_unaligned_access	3-207
3.182	--use_frame_pointer	3-209
3.183	--use_gas	3-210
3.184	--use_pch=filename	3-211
3.185	--using_std, --no_using_std	3-212
3.186	--vectorize, --no_vectorize	3-213
3.187	--version_number	3-214
3.188	--vfe, --no_vfe	3-215
3.189	--via=filename	3-216
3.190	--visibility_inlines_hidden	3-217
3.191	--vla, --no_vla	3-218
3.192	--vsn	3-219
3.193	-W	3-220
3.194	-Warmcc,option[,option,...]	3-221
3.195	-Warmcc,--gcc_fallback	3-222

3.196	--wchar, --no_wchar	3-223
3.197	--wchar16	3-224
3.198	--wchar32	3-225
3.199	--whole_program	3-226
3.200	--workdir=directory	3-227
3.201	--wrap_diagnostics, --no_wrap_diagnostics	3-228

Chapter 4

Language Extensions

4.1	Preprocessor extensions	4-3
4.2	#assert	4-4
4.3	#include_next	4-5
4.4	#unassert	4-6
4.5	#warning	4-7
4.6	C99 language features available in C90	4-8
4.7	// comments	4-9
4.8	Subscripting struct	4-10
4.9	Flexible array members	4-11
4.10	C99 language features available in C++ and C90	4-12
4.11	Variadic macros	4-13
4.12	long long	4-14
4.13	restrict	4-15
4.14	Hexadecimal floats	4-16
4.15	Standard C language extensions	4-17
4.16	Constant expressions	4-18
4.17	Array and pointer extensions	4-19
4.18	Block scope function declarations	4-20
4.19	Dollar signs in identifiers	4-21
4.20	Top-level declarations	4-22
4.21	Benign redeclarations	4-23
4.22	External entities	4-24
4.23	Function prototypes	4-25
4.24	Standard C++ language extensions	4-26
4.25	? operator	4-27
4.26	Declaration of a class member	4-28
4.27	friend	4-29
4.28	Read/write constants	4-30
4.29	Scalar type constants	4-32
4.30	Specialization of nonmember function templates	4-33
4.31	Type conversions	4-34
4.32	Standard C and Standard C++ language extensions	4-35
4.33	Address of a register variable	4-36
4.34	Arguments to functions	4-37
4.35	Anonymous classes, structures and unions	4-38
4.36	Assembler labels	4-39
4.37	Empty declaration	4-40
4.38	Hexadecimal floating-point constants	4-41
4.39	Incomplete enums	4-42
4.40	Integral type extensions	4-43
4.41	Label definitions	4-44
4.42	Long float	4-45
4.43	Nonstatic local variables	4-46
4.44	Structure, union, enum, and bitfield extensions	4-47
4.45	GNU extensions to the C and C++ languages	4-48

Chapter 5

Compiler-specific Features

5.1	Keywords and operators	5-5
5.2	__align	5-6
5.3	__alignof__	5-7
5.4	__ALIGNOF__	5-8
5.5	__asm	5-9

5.6	<code>__forceinline</code>	5-10
5.7	<code>__global_reg</code>	5-11
5.8	<code>__inline</code>	5-13
5.9	<code>__int64</code>	5-14
5.10	<code>__INTADDR__</code>	5-15
5.11	<code>__irq</code>	5-16
5.12	<code>__packed</code>	5-17
5.13	<code>__pure</code>	5-20
5.14	<code>__smc</code>	5-21
5.15	<code>__softfp</code>	5-22
5.16	<code>__svc</code>	5-23
5.17	<code>__svc_indirect</code>	5-24
5.18	<code>__svc_indirect_r7</code>	5-25
5.19	<code>__value_in_regs</code>	5-26
5.20	<code>__weak</code>	5-27
5.21	<code>__writeonly</code>	5-29
5.22	<code>__declspec</code> attributes	5-30
5.23	<code>__declspec(dllexport)</code>	5-31
5.24	<code>__declspec(dllimport)</code>	5-33
5.25	<code>__declspec(noinline)</code>	5-34
5.26	<code>__declspec(noreturn)</code>	5-35
5.27	<code>__declspec(nothrow)</code>	5-36
5.28	<code>__declspec(notshared)</code>	5-37
5.29	<code>__declspec(thread)</code>	5-38
5.30	Function attributes	5-39
5.31	<code>__attribute__((alias))</code> function attribute	5-41
5.32	<code>__attribute__((always_inline))</code> function attribute	5-42
5.33	<code>__attribute__((const))</code> function attribute	5-43
5.34	<code>__attribute__((constructor[<i>priority</i>]))</code> function attribute	5-44
5.35	<code>__attribute__((deprecated))</code> function attribute	5-46
5.36	<code>__attribute__((destructor[<i>priority</i>]))</code> function attribute	5-47
5.37	<code>__attribute__((format_arg(<i>string-index</i>)))</code> function attribute	5-48
5.38	<code>__attribute__((malloc))</code> function attribute	5-49
5.39	<code>__attribute__((noinline))</code> function attribute	5-50
5.40	<code>__attribute__((no_instrument_function))</code> function attribute	5-51
5.41	<code>__attribute__((nomerge))</code> function attribute	5-52
5.42	<code>__attribute__((nonnull))</code> function attribute	5-53
5.43	<code>__attribute__((noreturn))</code> function attribute	5-54
5.44	<code>__attribute__((notailcall))</code> function attribute	5-55
5.45	<code>__attribute__((pcs("calling_convention")))</code>	5-56
5.46	<code>__attribute__((pure))</code> function attribute	5-57
5.47	<code>__attribute__((section("name")))</code> function attribute	5-58
5.48	<code>__attribute__((unused))</code> function attribute	5-59
5.49	<code>__attribute__((used))</code> function attribute	5-60
5.50	<code>__attribute__((visibility("visibility_type")))</code> function attribute	5-61
5.51	<code>__attribute__((weak))</code> function attribute	5-62
5.52	<code>__attribute__((weakref("target")))</code> function attribute	5-63
5.53	Type attributes	5-64
5.54	<code>__attribute__((bitband))</code> type attribute	5-65
5.55	<code>__attribute__((aligned))</code> type attribute	5-67
5.56	<code>__attribute__((packed))</code> type attribute	5-68
5.57	<code>__attribute__((transparent_union))</code> type attribute	5-69
5.58	Variable attributes	5-70
5.59	<code>__attribute__((alias))</code> variable attribute	5-71
5.60	<code>__attribute__((at(<i>address</i>)))</code> variable attribute	5-72
5.61	<code>__attribute__((aligned))</code> variable attribute	5-73
5.62	<code>__attribute__((deprecated))</code> variable attribute	5-74
5.63	<code>__attribute__((noinline))</code> constant variable attribute	5-75
5.64	<code>__attribute__((packed))</code> variable attribute	5-76
5.65	<code>__attribute__((section("name")))</code> variable attribute	5-77

5.66	__attribute__((transparent_union)) variable attribute	5-78
5.67	__attribute__((unused)) variable attribute	5-79
5.68	__attribute__((used)) variable attribute	5-80
5.69	__attribute__((visibility("visibility_type"))) variable attribute	5-81
5.70	__attribute__((weak)) variable attribute	5-82
5.71	__attribute__((weakref("target"))) variable attribute	5-83
5.72	__attribute__((zero_init)) variable attribute	5-84
5.73	Pragmas	5-85
5.74	#pragma anon_unions, #pragma no_anon_unions	5-86
5.75	#pragma arm	5-87
5.76	#pragma arm section [section_type_list]	5-88
5.77	#pragma diag_default tag[,tag,...]	5-90
5.78	#pragma diag_error tag[,tag,...]	5-91
5.79	#pragma diag_remark tag[,tag,...]	5-92
5.80	#pragma diag_suppress tag[,tag,...]	5-93
5.81	#pragma diag_warning tag[, tag, ...]	5-94
5.82	#pragma exceptions_unwind, #pragma no_exceptions_unwind	5-95
5.83	#pragma GCC system_header	5-96
5.84	#pragma hdrstop	5-97
5.85	#pragma import symbol_name	5-98
5.86	#pragma import(__use_full_stdio)	5-99
5.87	#pragma import(__use_smaller_memcpy)	5-100
5.88	#pragma inline, #pragma no_inline	5-101
5.89	#pragma no_pch	5-102
5.90	#pragma Onum	5-103
5.91	#pragma once	5-104
5.92	#pragma Ospace	5-105
5.93	#pragma Otime	5-106
5.94	#pragma pack(n)	5-107
5.95	#pragma pop	5-109
5.96	#pragma push	5-110
5.97	#pragma softfp_linkage, #pragma no_softfp_linkage	5-111
5.98	#pragma thumb	5-112
5.99	#pragma unroll [(n)]	5-113
5.100	#pragma unroll_completely	5-115
5.101	#pragma weak symbol, #pragma weak symbol1 = symbol2	5-116
5.102	Instruction intrinsics	5-117
5.103	__breakpoint intrinsic	5-118
5.104	__cdp intrinsic	5-119
5.105	__clrex intrinsic	5-120
5.106	__clz intrinsic	5-121
5.107	__current_pc intrinsic	5-122
5.108	__current_sp intrinsic	5-123
5.109	__disable_fiq intrinsic	5-124
5.110	__disable_irq intrinsic	5-125
5.111	__enable_fiq intrinsic	5-127
5.112	__enable_irq intrinsic	5-128
5.113	__fabs intrinsic	5-129
5.114	__fabsf intrinsic	5-130
5.115	__force_stores intrinsic	5-131
5.116	__ldrex intrinsic	5-132
5.117	__ldrexld intrinsic	5-134
5.118	__ldrt intrinsic	5-135
5.119	__memory_changed intrinsic	5-137
5.120	__nop intrinsic	5-138
5.121	__pld intrinsic	5-141
5.122	__pldw intrinsic	5-142
5.123	__pli intrinsic	5-143
5.124	__promise intrinsic	5-144
5.125	__qadd intrinsic	5-145

5.126	__qdbl intrinsic	5-146
5.127	__qsub intrinsic	5-147
5.128	__rbit intrinsic	5-148
5.129	__rev intrinsic	5-149
5.130	__return_address intrinsic	5-150
5.131	__ror intrinsic	5-151
5.132	__schedule_barrier intrinsic	5-152
5.133	__semihost intrinsic	5-153
5.134	__sev intrinsic	5-155
5.135	__sqrt intrinsic	5-156
5.136	__sqrtf intrinsic	5-157
5.137	__ssat intrinsic	5-158
5.138	__strex intrinsic	5-159
5.139	__strexld intrinsic	5-161
5.140	__strt intrinsic	5-162
5.141	__swp intrinsic	5-163
5.142	__usat intrinsic	5-164
5.143	__wfe intrinsic	5-165
5.144	__wfi intrinsic	5-166
5.145	__yield intrinsic	5-167
5.146	ARMv6 SIMD intrinsics	5-168
5.147	ETSI basic operations	5-169
5.148	C55x intrinsics	5-171
5.149	VFP status intrinsic	5-173
5.150	__vfp_status intrinsic	5-174
5.151	Fused Multiply Add (FMA) intrinsics	5-175
5.152	Named register variables	5-176
5.153	GNU built-in functions	5-180
5.154	Predefined macros	5-183
5.155	Built-in function name variables	5-189

Chapter 6

C and C++ Implementation Details

6.1	C and C++ implementation details	6-2
6.2	Character sets and identifiers	6-3
6.3	Basic data types	6-5
6.4	Operations on basic data types	6-7
6.5	Structures, unions, enumerations, and bitfields	6-9
6.6	C++ implementation details	6-14
6.7	Using the ::operator new function	6-15
6.8	Tentative arrays	6-16
6.9	Old-style C parameters in C++ functions	6-17
6.10	Anachronisms	6-18
6.11	Template instantiation	6-19
6.12	Namespaces	6-20
6.13	C++ exception handling	6-22
6.14	Extern inline functions	6-23

Appendix A

ARMv6 SIMD Instruction Intrinsics

A.1	ARMv6 SIMD intrinsics by prefix	A-3
A.2	ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags	A-5
A.3	ARMv6 SIMD intrinsics, compatible processors and architectures	A-9
A.4	ARMv6 SIMD instruction intrinsics and APSR GE flags	A-10
A.5	__qadd16 intrinsic	A-11
A.6	__qadd8 intrinsic	A-12
A.7	__qasx intrinsic	A-13
A.8	__qsax intrinsic	A-14
A.9	__qsub16 intrinsic	A-15
A.10	__qsub8 intrinsic	A-16
A.11	__sadd16 intrinsic	A-17
A.12	__sadd8 intrinsic	A-18

A.13	__sasx intrinsic	A-19
A.14	__sel intrinsic	A-20
A.15	__shadd16 intrinsic	A-21
A.16	__shadd8 intrinsic	A-22
A.17	__shasx intrinsic	A-23
A.18	__shsax intrinsic	A-24
A.19	__shsub16 intrinsic	A-25
A.20	__shsub8 intrinsic	A-26
A.21	__smlad intrinsic	A-27
A.22	__smladx intrinsic	A-28
A.23	__smlald intrinsic	A-29
A.24	__smlaldx intrinsic	A-30
A.25	__smlsd intrinsic	A-31
A.26	__smlsdx intrinsic	A-32
A.27	__smlsld intrinsic	A-33
A.28	__smlsldx intrinsic	A-34
A.29	__smuad intrinsic	A-35
A.30	__smuadx intrinsic	A-36
A.31	__smusd intrinsic	A-37
A.32	__smusdx intrinsic	A-38
A.33	__ssat16 intrinsic	A-39
A.34	__ssax intrinsic	A-40
A.35	__ssub16 intrinsic	A-41
A.36	__ssub8 intrinsic	A-42
A.37	__sxtab16 intrinsic	A-43
A.38	__sxtb16 intrinsic	A-44
A.39	__uadd16 intrinsic	A-45
A.40	__uadd8 intrinsic	A-46
A.41	__uasx intrinsic	A-47
A.42	__uhadd16 intrinsic	A-48
A.43	__uhadd8 intrinsic	A-49
A.44	__uhasx intrinsic	A-50
A.45	__uhsax intrinsic	A-51
A.46	__uhsub16 intrinsic	A-52
A.47	__uhsub8 intrinsic	A-53
A.48	__uqadd16 intrinsic	A-54
A.49	__uqadd8 intrinsic	A-55
A.50	__uqasx intrinsic	A-56
A.51	__uqsax intrinsic	A-57
A.52	__uqsub16 intrinsic	A-58
A.53	__uqsub8 intrinsic	A-59
A.54	__usad8 intrinsic	A-60
A.55	__usada8 intrinsic	A-61
A.56	__usax intrinsic	A-62
A.57	__usat16 intrinsic	A-63
A.58	__usub16 intrinsic	A-64
A.59	__usub8 intrinsic	A-65
A.60	__uxtab16 intrinsic	A-66
A.61	__uxtb16 intrinsic	A-67

Appendix B**Via File Syntax**

B.1	Overview of via files	B-2
B.2	Via file syntax	B-3

Appendix C**Summary Table of GNU Language Extensions****Appendix D****Standard C Implementation Definition**

D.1	Implementation definition	D-2
D.2	Translation	D-3
D.3	Environment	D-4

D.4	Identifiers	D-6
D.5	Characters	D-7
D.6	Integers	D-8
D.7	Floating-point	D-9
D.8	Arrays and pointers	D-10
D.9	Registers	D-11
D.10	Structures, unions, enumerations, and bitfields	D-12
D.11	Qualifiers	D-13
D.12	Expression evaluation	D-14
D.13	Preprocessing directives	D-15
D.14	Library functions	D-16
D.15	Behaviors considered undefined by the ISO C Standard	D-17
Appendix E	Standard C++ Implementation Definition	
E.1	Integral conversion	E-2
E.2	Calling a pure virtual function	E-3
E.3	Major features of language support	E-4
E.4	Standard C++ library implementation definition	E-5
Appendix F	C and C++ Compiler Implementation Limits	
F.1	C++ ISO/IEC standard limits	F-2
F.2	Limits for integral numbers	F-4
F.3	Limits for floating-point numbers	F-5
Appendix G	Using NEON Support	
G.1	Introduction	G-3
G.2	Vector data types	G-4
G.3	Summary of NEON intrinsics	G-5
G.4	Intrinsics	G-10
G.5	Addition	G-12
G.6	Multiplication	G-14
G.7	Subtraction	G-16
G.8	Comparison	G-18
G.9	Absolute difference	G-21
G.10	Max/Min	G-22
G.11	Pairwise addition	G-23
G.12	Folding maximum	G-24
G.13	Folding minimum	G-25
G.14	Reciprocal/Sqrt	G-26
G.15	Shifts by signed variable	G-27
G.16	Shifts by a constant	G-29
G.17	Shifts with insert	G-33
G.18	Loads of a single vector or lane	G-34
G.19	Store a single vector or lane	G-37
G.20	Loads of an N-element structure	G-39
G.21	Extract lanes from a vector and put into a register	G-48
G.22	Load a single lane of a vector from a literal	G-49
G.23	Initialize a vector from a literal bit pattern	G-50
G.24	Set all lanes to same value	G-51
G.25	Combining vectors	G-53
G.26	Splitting vectors	G-54
G.27	Converting vectors	G-55
G.28	Table look up	G-57
G.29	Extended table look up intrinsics	G-58
G.30	Operations with a scalar value	G-59
G.31	Vector extract	G-64
G.32	Reverse vector elements (swap endianness)	G-65
G.33	Other single operand arithmetic	G-66
G.34	Logical operations	G-68
G.35	Transposition operations	G-71

G.36 Vector reinterpret cast operations G-73

Appendix H Revisions for the Compiler Reference

Chapter 1

Conventions and Feedback

The following describes the typographical conventions and how to give feedback:

Typographical conventions

The following typographical conventions are used:

`monospace` Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

monospace Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

monospace italic

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM[®] processor signal names.

Feedback on this product

If you have any comments and suggestions about this product, contact your supplier and give:

- your name and company

- the serial number of the product
- details of the release you are using
- details of the platform you are using, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tools, including the version number and build numbers.

Feedback on documentation

If you have comments on the documentation, e-mail errata@arm.com. Give:

- the title
- the number, ARM DUI 0491H
- if viewing online, the topic names to which your comments apply
- if viewing a PDF version of a document, the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

ARM periodically provides updates and corrections to its documentation on the ARM Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

Other information

- ARM Information Center, <http://infocenter.arm.com/help/index.jsp>
- ARM Technical Support Knowledge Articles, <http://infocenter.arm.com/help/topic/com.arm.doc.faqs/index.html>
- ARM Support and Maintenance, <http://www.arm.com/support/services/support-maintenance.php>
- ARM Glossary, <http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html>.

Chapter 2

Introduction

The following topics introduce the compiler, armcc:

- *About the ARM compiler* on page 2-2
- *Source language modes of the compiler* on page 2-3
 - *ISO C90* on page 2-4
 - *ISO C99* on page 2-5
 - *ISO C++* on page 2-6
- *Language extensions and language compliance* on page 2-7
 - *Language extensions* on page 2-8
 - *Language compliance* on page 2-9
- *The C and C++ libraries* on page 2-10.

2.1 About the ARM compiler

The compiler, `armcc`, enables you to compile your C and C++ code.

The compiler:

- Is an optimizing compiler. Command-line options enable you to control the level of optimization.
- Compiles:
 - ISO Standard C:1990 source
 - ISO Standard C:1999 source
 - ISO Standard C++:2003 source.into:
 - 32-bit ARM code
 - 16/32-bit Thumb® code
 - 16-bit Thumb code.
- Complies with the *Base Standard Application Binary Interface for the ARM Architecture (BSABI)*. In particular, the compiler:
 - generates output objects in ELF format
 - generates DWARF Debugging Standard Version 3 (DWARF 3) debug information and contains support for DWARF 2 debug tables.See [Compliance with the Application Binary Interface \(ABI\) for the ARM architecture on page 2-9](#) in *Using ARM C and C++ Libraries and Floating-Point Support* for more information.
- Can generate an assembly language listing of the output code, and can interleave an assembly language listing with source code.

2.2 Source language modes of the compiler

The compiler has three distinct source language modes that you can use to compile different varieties of C and C++ source code. These are:

- ISO C90
- ISO C99
- ISO C++.

See:

- [ISO C90 on page 2-4](#)
- [ISO C99 on page 2-5](#)
- [ISO C++ on page 2-6](#).

2.3 ISO C90

The compiler compiles C as defined by the 1990 C standard and addenda:

- ISO/IEC 9899:1990. The 1990 International Standard for C.
- ISO/IEC 9899 AM1. The 1995 Normative Addendum 1, adding international character support through `wchar.h` and `wtype.h`.

The compiler also supports several extensions to ISO C90. See [Language extensions and language compliance on page 2-7](#) for more information.

Throughout this document, the term:

C90 Means ISO C90, together with the ARM extensions.
Use the compiler option `--c90` to compile C90 code. This is the default.

Strict C90 Means C as defined by the 1990 C standard and addenda.

2.3.1 See also

- [--c90 on page 3-33](#)
- [--strict, --no_strict on page 3-194](#)
- [Language extensions and language compliance on page 2-7](#)
- [Appendix D Standard C Implementation Definition.](#)

2.4 ISO C99

The compiler compiles C as defined by the 1999 C standard and addenda:

- ISO/IEC 9899:1999. The 1999 International Standard for C.
- ISO/IEC 9899:1999/Cor 2:2004. Technical Corrigendum 2.

The compiler also supports several extensions to ISO C99. See *Language extensions and language compliance on page 2-7* for more information.

Throughout this document, the term:

C99 Means ISO C99, together with the ARM and GNU extensions.
Use the compiler option `--c99` to compile C99 code.

Strict C99 Means C as defined by the 1999 C standard and addenda.

Standard C Means C90 or C99 as appropriate.

C Means any of C90, strict C90, C99, and Standard C.

2.4.1 See also

- [--c99 on page 3-34](#)
- [--strict, --no_strict on page 3-194](#)
- [Language extensions and language compliance on page 2-7](#)
- [Appendix D Standard C Implementation Definition.](#)

2.5 ISO C++

The compiler compiles C++ as defined by the 2003 standard, excepting wide streams and export templates:

- ISO/IEC 14822:2003. The 2003 International Standard for C++.

The compiler also supports several extensions to ISO C++. See [Language extensions and language compliance on page 2-7](#) for more information.

Throughout this document, the term:

strict C++ Means ISO C++, excepting wide streams and export templates.

Standard C++ Means strict C++.

C++ Means ISO C++, excepting wide streams and export templates, either with or without the ARM extensions.

Use the compiler option `--cpp` to compile C++ code.

2.5.1 See also

- [--cpp on page 3-47](#)
- [--strict, --no_strict on page 3-194](#)
- [Language extensions and language compliance on page 2-7](#)
- [Appendix E Standard C++ Implementation Definition.](#)

2.6 Language extensions and language compliance

The compiler supports numerous extensions to its various source languages. It also provides several command-line options for controlling compliance with the available source languages.

See:

- [Language extensions on page 2-8](#)
- [Language compliance on page 2-9](#).

2.7 Language extensions

The language extensions supported by the compiler are categorized as follows:

- C99 features** The compiler makes some language features of C99 available:
- as extensions to strict C90, for example, `//-style comments`
 - as extensions to both Standard C++ and strict C90, for example, **restrict** pointers.
- For more information see:
- [C99 language features available in C90 on page 4-8](#)
 - [C99 language features available in C++ and C90 on page 4-12.](#)

Standard C extensions

The compiler supports numerous extensions to strict C99, for example, function prototypes that override old-style nonprototype definitions. See [Standard C language extensions on page 4-17](#) for more information.

These extensions to Standard C are also available in C90.

Standard C++ extensions

The compiler supports numerous extensions to strict C++, for example, qualified names in the declaration of class members. See [Standard C++ language extensions on page 4-26](#) for more information.

These extensions are not available in either Standard C or C90.

Standard C and Standard C++ extensions

The compiler supports some extensions specific to strict C++ and strict C90, for example, anonymous classes, structures, and unions. See [Standard C and Standard C++ language extensions on page 4-35](#) for more information.

- GNU extensions** The compiler supports some extensions offered by the GNU compiler. See:
- [Language compliance on page 2-9](#)
 - [GNU extensions to the C and C++ languages on page 4-48](#)
 - [Chapter 5 Compiler-specific Features.](#)

ARM-specific extensions

The compiler supports a range of extensions specific to the ARM compiler, for example, instruction intrinsics and other built-in functions. See [Chapter 5 Compiler-specific Features](#) for more information.

2.8 Language compliance

The compiler has several modes where compliance to a source language is either enforced or relaxed:

Strict mode In strict mode the compiler enforces compliance with the language standard relevant to the source language. For example, the use of `//`-style comments results in an error when compiling strict C90.

To compile in strict mode, use the command-line option `--strict`.

GNU mode In GNU mode all the GNU compiler extensions to the relevant source language are available. For example, in GNU mode:

- case ranges in `switch` statements are available when the source language is any of C90, C99 or nonstrict C++
- C99-style designated initializers are available when the source language is either C90 or nonstrict C++.

To compile in GNU mode, use the compiler option `--gnu`.

———— **Note** —————

Some GNU extensions are also available when you are in a nonstrict mode.

2.8.1 Example

The following examples illustrate combining source language modes with language compliance modes:

- Compiling a `.cpp` file with the command-line option `--strict` compiles Standard C++
- Compiling a C source file with the command-line option `--gnu` compiles GNU mode C90
- Compiling a `.c` file with the command-line options `--strict` and `--gnu` is an error.

2.8.2 See also

- [--gnu on page 3-107](#)
- [--strict, --no_strict on page 3-194](#)
- [GNU extensions to the C and C++ languages on page 4-48](#)
- [Filename suffixes recognized by the compiler on page 3-15](#) in *Using the Compiler*.

2.9 The C and C++ libraries

The following runtime C and C++ libraries are provided:

The ARM C libraries

The ARM C libraries provide standard C functions, and helper functions used by the C and C++ libraries. The C libraries also provide target-dependent functions that are used to implement the standard C library functions such as `printf` in a semihosted environment. The C libraries are structured so that you can redefine target-dependent functions in your own code to remove semihosting dependencies.

The ARM libraries comply with:

- the *C Library ABI for the ARM Architecture* (CLIBABI)
- the *C++ ABI for the ARM Architecture* (CPPABI).

See *Compliance with the Application Binary Interface (ABI) for the ARM architecture on page 2-9* in *Using ARM C and C++ Libraries and Floating-Point Support* for more information.

Rogue Wave Standard C++ Library version 2.02.03

The Rogue Wave Standard C++ Library, as supplied by Rogue Wave Software, Inc., provides Standard C++ functions and objects such as `cout`. It includes data structures and algorithms known as the *Standard Template Library* (STL). The C++ libraries use the C libraries to provide target-specific support. The Rogue Wave Standard C++ Library is provided with C++ exceptions enabled.

For more information on the Rogue Wave libraries, see the Rogue Wave HTML documentation and the Rogue Wave web site at: <http://www.roguewave.com>

Support libraries

The ARM C libraries provide additional components to enable support for C++ and to compile code for different architectures and processors.

The C and C++ libraries are provided as binaries only. There is a variant of the 1990 ISO Standard C library for each combination of major build options, such as the byte order of the target system, whether interworking is selected, and whether floating-point support is selected.

See *Chapter 2 The ARM C and C++ libraries* in *Using ARM C and C++ Libraries and Floating-Point Support* for more information.

Chapter 3

Compiler Command-line Options

The following command-line options are accepted by the compiler, `armcc`, version 4.1 and later:

- `-Aopt` on page 3-6
- `--allow_fpreg_for_nonfpdata`, `--no_allow_fpreg_for_nonfpdata` on page 3-7
- `--allow_null_this`, `--no_allow_null_this` on page 3-8
- `--alternative_tokens`, `--no_alternative_tokens` on page 3-9
- `--anachronisms`, `--no_anachronisms` on page 3-10
- `--apcs=qualifer...qualifier` on page 3-11
- `--arm` on page 3-15
- `--arm_linux` on page 3-16
- `--arm_linux_config_file=path` on page 3-18
- `--arm_linux_configure` on page 3-19
- `--arm_linux_paths` on page 3-21
- `--arm_only` on page 3-23
- `--asm` on page 3-24
- `--asm_dir=directory_name` on page 3-25
- `--autoinline`, `--no_autoinline` on page 3-26
- `--bigend` on page 3-27
- `--bitband` on page 3-28
- `--brief_diagnostics`, `--no_brief_diagnostics` on page 3-29
- `--bss_threshold=num` on page 3-30
- `-c` on page 3-31
- `-C` on page 3-32
- `--c90` on page 3-33

- `--c99` on page 3-34
- `--code_gen`, `--no_code_gen` on page 3-35
- `--compatible=name` on page 3-36
- `--compile_all_input`, `--no_compile_all_input` on page 3-37
- `--conditionalize`, `--no_conditionalize` on page 3-38
- `--configure_cpp_headers=path` on page 3-39
- `--configure_extra_includes=paths` on page 3-40
- `--configure_extra_libraries=paths` on page 3-41
- `--configure_gas=path` on page 3-42
- `--configure_gcc=path` on page 3-43
- `--configure_gcc_version=version` on page 3-44
- `--configure_gld=path` on page 3-45
- `--configure_sysroot=path` on page 3-46
- `--cpp` on page 3-47
- `--cpu=list` on page 3-48
- `--cpu=name` on page 3-49
- `--create_pch=filename` on page 3-53
- `-Dname[(parm-list)][=def]` on page 3-54
- `--data_reorder`, `--no_data_reorder` on page 3-55
- `--debug`, `--no_debug` on page 3-56
- `--debug_macros`, `--no_debug_macros` on page 3-57
- `--default_definition_visibility=visibility` on page 3-58
- `--default_extension=ext` on page 3-59
- `--dep_name`, `--no_dep_name` on page 3-60
- `--depend=filename` on page 3-61
- `--depend_dir=directory_name` on page 3-62
- `--depend_format=string` on page 3-63
- `--depend_single_line`, `--no_depend_single_line` on page 3-65
- `--depend_system_headers`, `--no_depend_system_headers` on page 3-66
- `--depend_target=target` on page 3-67
- `--device=list` on page 3-68
- `--device=name` on page 3-69
- `--diag_error=tag[,tag,...]` on page 3-70
- `--diag_remark=tag[,tag,...]` on page 3-71
- `--diag_style={arm|ide|gnu}` on page 3-72
- `--diag_suppress=tag[,tag,...]` on page 3-73
- `--diag_suppress=optimizations` on page 3-74
- `--diag_warning=tag[,tag,...]` on page 3-75
- `--diag_warning=optimizations` on page 3-76
- `--dllexport_all`, `--no_dllexport_all` on page 3-77
- `--dllimport_runtime`, `--no_dllimport_runtime` on page 3-78
- `--dollar`, `--no_dollar` on page 3-79
- `--dwarf2` on page 3-80
- `--dwarf3` on page 3-81
- `-E` on page 3-82
- `--echo` on page 3-83
- `--emit_frame_directives`, `--no_emit_frame_directives` on page 3-84

- `--enum_is_int` on page 3-85
- `--errors=filename` on page 3-86
- `--exceptions`, `--no_exceptions` on page 3-87
- `--exceptions_unwind`, `--no_exceptions_unwind` on page 3-88
- `--execstack`, `--no_execstack` on page 3-89
- `--export_all_vtbl`, `--no_export_all_vtbl` on page 3-90
- `--export_defs_implicitly`, `--no_export_defs_implicitly` on page 3-91
- `--extended_initializers`, `--no_extended_initializers` on page 3-92
- `--feedback=filename` on page 3-93
- `--force_new_nothrow`, `--no_force_new_nothrow` on page 3-94
- `--forceinline` on page 3-95
- `--fp16_format=format` on page 3-96
- `--fpmode=model` on page 3-97
- `--fpu=list` on page 3-99
- `--fpu=name` on page 3-100
- `--friend_injection`, `--no_friend_injection` on page 3-104
- `-g` on page 3-105
- `--global_reg=reg_name[,reg_name,...]` on page 3-106
- `--gnu` on page 3-107
- `--gnu_defaults` on page 3-108
- `--gnu_instrument`, `--no_gnu_instrument` on page 3-109
- `--gnu_version=version` on page 3-110
- `--guiding_decls`, `--no_guiding_decls` on page 3-111
- `--help` on page 3-112
- `--hide_all`, `--no_hide_all` on page 3-113
- `-Idir[,dir,...]` on page 3-114
- `--ignore_missing_headers` on page 3-115
- `--implicit_include`, `--no_implicit_include` on page 3-116
- `--implicit_include_searches`, `--no_implicit_include_searches` on page 3-117
- `--implicit_key_function`, `--no_implicit_key_function` on page 3-118
- `--implicit_typename`, `--no_implicit_typename` on page 3-119
- `--import_all_vtbl` on page 3-120
- `--info=totals` on page 3-121
- `--inline`, `--no_inline` on page 3-122
- `--interface_enums_are_32_bit` on page 3-123
- `--interleave` on page 3-124
- `-Jdir[,dir,...]` on page 3-125
- `--kandr_include` on page 3-126
- `-Lopt` on page 3-127
- `--library_interface=lib` on page 3-128
- `--library_type=lib` on page 3-130
- `--licrety` on page 3-131
- `--link_all_input`, `--no_link_all_input` on page 3-132
- `--list` on page 3-133
- `--list_dir=directory_name` on page 3-135
- `--list_macros` on page 3-136
- `--littleend` on page 3-137

- `--locale=lang_country` on page 3-138
- `--long_long` on page 3-139
- `--loose_implicit_cast` on page 3-140
- `--lower_ropi, --no_lower_ropi` on page 3-141
- `--lower_rwpi, --no_lower_rwpi` on page 3-142
- `--ltcg` on page 3-143
- `-M` on page 3-144
- `--md` on page 3-145
- `--message_locale=lang_country[.codepage]` on page 3-146
- `--min_array_alignment=opt` on page 3-147
- `--mm` on page 3-148
- `--multibyte_chars, --no_multibyte_chars` on page 3-149
- `--multifile, --no_multifile` on page 3-150
- `--multiply_latency=cycles` on page 3-151
- `--narrow_volatile_bitfields` on page 3-152
- `--nonstd_qualifier_deduction, --no_nonstd_qualifier_deduction` on page 3-153
- `-o filename` on page 3-154
- `-Onum` on page 3-156
- `--old_specializations, --no_old_specializations` on page 3-158
- `--old_style_preprocessing` on page 3-159
- `-Ospace` on page 3-160
- `-Otime` on page 3-161
- `--output_dir=directory_name` on page 3-162
- `-P` on page 3-163
- `--parse_templates, --no_parse_templates` on page 3-164
- `--pch` on page 3-165
- `--pch_dir=dir` on page 3-166
- `--pch_messages, --no_pch_messages` on page 3-167
- `--pch_verbose, --no_pch_verbose` on page 3-168
- `--pending_instantiations=n` on page 3-169
- `--phony_targets` on page 3-170
- `--pointer_alignment=num` on page 3-171
- `--preinclude=filename` on page 3-172
- `--preprocess_assembly` on page 3-173
- `--preprocessed` on page 3-174
- `--project=filename, --no_project` on page 3-175
- `--protect_stack, --no_protect_stack` on page 3-176
- `--reassociate_saturation, --no_reassociate_saturation` on page 3-177
- `--reduce_paths, --no_reduce_paths` on page 3-178
- `--reinitialize_workdir` on page 3-179
- `--relaxed_ref_def, --no_relaxed_ref_def` on page 3-180
- `--remarks` on page 3-181
- `--remove_unneeded_entities, --no_remove_unneeded_entities` on page 3-182
- `--restrict, --no_restrict` on page 3-183
- `--retain=option` on page 3-184
- `--rtti, --no_rtti` on page 3-185
- `--rtti_data, --no_rtti_data` on page 3-186

- *-S* on page 3-187
- *--shared* on page 3-188
- *--show_cmdline* on page 3-189
- *--signed_bitfields*, *--unsigned_bitfields* on page 3-190
- *--signed_chars*, *--unsigned_chars* on page 3-191
- *--split_ldm* on page 3-192
- *--split_sections* on page 3-193
- *--strict*, *--no_strict* on page 3-194
- *--strict_warnings* on page 3-195
- *--sys_include* on page 3-196
- *--thumb* on page 3-197
- *--translate_g++* on page 3-198
- *--translate_gcc* on page 3-200
- *--translate_gld* on page 3-202
- *--trigraphs*, *--no_trigraphs* on page 3-204
- *--type_traits_helpers*, *--no_type_traits_helpers* on page 3-205
- *-Uname* on page 3-206
- *--unaligned_access*, *--no_unaligned_access* on page 3-207
- *--use_frame_pointer* on page 3-209
- *--use_gas* on page 3-210
- *--use_pch=filename* on page 3-211
- *--using_std*, *--no_using_std* on page 3-212
- *--vectorize*, *--no_vectorize* on page 3-213
- *--version_number* on page 3-214
- *--vfe*, *--no_vfe* on page 3-215
- *--via=filename* on page 3-216
- *--visibility_inlines_hidden* on page 3-217
- *--vla*, *--no_vla* on page 3-218
- *--vsn* on page 3-219
- *-W* on page 3-220
- *-Warmcc,option[,option,...]* on page 3-221
- *-Warmcc,--gcc_fallback* on page 3-222
- *--wchar*, *--no_wchar* on page 3-223
- *--wchar16* on page 3-224
- *--wchar32* on page 3-225
- *--whole_program* on page 3-226
- *--workdir=directory* on page 3-227
- *--wrap_diagnostics*, *--no_wrap_diagnostics* on page 3-228.

3.1 *-Aopt*

This option specifies command-line options to pass to the assembler when it is invoked by the compiler to assemble either `.s` input files or embedded assembly language functions.

3.1.1 Syntax

-Aopt

Where:

opt is a command-line option to pass to the assembler.

———— **Note** —————

Some compiler command-line options are passed to the assembler automatically whenever it is invoked by the compiler. For example, if the option `--cpu` is specified on the compiler command line, then this option is passed to the assembler whenever it is invoked to assemble `.s` files or embedded assembler.

To see the compiler command-line options passed by the compiler to the assembler, use the compiler command-line option `-A--show_cmdline`.

3.1.2 Example

```
armcc -A--predefine="NEWVERSION SETL {TRUE}" main.c
```

3.1.3 Restrictions

If an unsupported option is passed through using `-A`, an error is generated by the assembler.

3.1.4 See also

- [--cpu=name](#) on page 3-49
- [-Lopt](#) on page 3-127
- [--show_cmdline](#) on page 3-189.

3.2 `--allow_fpreg_for_nonfpdata`, `--no_allow_fpreg_for_nonfpdata`

These options enable or disable the use of VFP and NEON registers and data transfer instructions for non-VFP and non-NEON data.

3.2.1 Usage

`--allow_fpreg_for_nonfpdata` enables the compiler to use VFP and NEON registers and instructions for data transfer operations on non-VFP and non-NEON data. This is useful when demand for integer registers is high. For the compiler to use the VFP or NEON registers, the default options for the processor or the specified options must enable the hardware.

`--no_allow_fpreg_for_nonfpdata` prevents VFP and NEON registers from being used for non-VFP and non-NEON data. When this option is specified, the compiler uses VFP and NEON registers for VFP and NEON data only. This is useful when you want to confine the number of places in your code where the compiler generates VFP or NEON instructions.

3.2.2 Default

The default is `--no_allow_fpreg_for_nonfpdata`.

3.2.3 See also

- [--fpmode=model](#) on page 3-97
- [--fpu=list](#) on page 3-99
- [--fpu=name](#) on page 3-100
- [Extension register bank mapping on page 9-6](#) in Using the Assembler
- [NEON views of the register bank on page 9-8](#) in Using the Assembler
- [VFP views of the extension register bank on page 9-9](#) in Using the Assembler.

3.3 `--allow_null_this`, `--no_allow_null_this`

These options allow and disallow null **this** pointers in C++.

3.3.1 Usage

Allowing null **this** pointers gives well-defined behavior when a nonvirtual member function is called on a null object pointer.

Disallowing null **this** pointers enables the compiler to perform optimizations, and conforms with the C++ standard.

3.3.2 Default

The default is `--no_allow_null_this`.

3.3.3 See also

- [--gnu_defaults](#) on page 3-108.

3.4 --alternative_tokens, --no_alternative_tokens

This option enables or disables the recognition of alternative tokens in C and C++.

3.4.1 Usage

In C and C++, use this option to control recognition of the digraphs. In C++, use this option to control recognition of operator keywords, for example, **and** and **bitand**.

3.4.2 Default

The default is --alternative_tokens.

3.5 --anachronisms, --no_anachronisms

This option enables or disables anachronisms in C++.

3.5.1 Mode

This option is effective only if the source language is C++.

3.5.2 Default

The default is `--no_anachronisms`.

3.5.3 Example

```
typedef enum { red, white, blue } tricolor;
inline tricolor operator++(tricolor c, int)
{
    int i = static_cast<int>(c) + 1;
    return static_cast<tricolor>(i);
}
void foo(void)
{
    tricolor c = red;
    c++; // okay
    ++c; // anachronism
}
```

Compiling this code with the option `--anachronisms` generates a warning message.

Compiling this code without the option `--anachronisms` generates an error message.

3.5.4 See also

- [--cpp on page 3-47](#)
- [--strict, --no_strict on page 3-194](#)
- [--strict_warnings on page 3-195](#)
- [Anachronisms on page 6-18.](#)

3.6 `--apcs=qualifer...qualifier`

This option controls interworking and position independence when generating code.

By specifying qualifiers to the `--apcs` command-line option, you can define the variant of the *Procedure Call Standard for the ARM architecture* (AAPCS) used by the compiler.

3.6.1 Syntax

`--apcs=qualifer...qualifier`

Where *qualifer...qualifier* denotes a list of qualifiers. There must be:

- at least one qualifier present
- no spaces separating individual qualifiers in the list.

Each instance of *qualifier* must be one of:

/interwork, /nointerwork

Generates code with or without ARM/Thumb interworking support. The default is `/nointerwork`, except for ARMv5T and later where the default is `/interwork`.

/ropi, /noropi

Enables or disables the generation of *Read-Only Position-Independent* (ROPI) code. The default is `/noropi`.

`/[no]pic` is an alias for `/[no]ropi`.

/rwpi, /norwpi

Enables or disables the generation of *Read/Write Position-Independent* (RWPI) code. The default is `/norwpi`.

`/[no]pid` is an alias for `/[no]rwpi`.

/fpic, /nofpic

Enables or disables the generation of read-only position-independent code where relative address references are independent of the location where your program is loaded.

/hardfp, /softfp

Requests hardware or software floating-point linkage. This enables the procedure call standard to be specified separately from the version of the floating-point hardware available through the `--fpu` option. It is still possible to specify the procedure call standard by using the `--fpu` option, but the use of `--apcs` is recommended.

Note

You can alternatively specify multiple qualifiers. For example,

`--apcs=/nointerwork/noropi/norwpi` is equivalent to `--apcs=/nointerwork`

`--apcs=noropi/norwpi`.

3.6.2 Default

If you do not specify an `--apcs` option, the compiler assumes

`--apcs=/nointerwork/noropi/norwpi/nofpic`.

3.6.3 Usage

/interwork, /nointerwork

By default, code is generated:

- without interworking support, that is /nointerwork, unless you specify a --cpu option that corresponds to architecture ARMv5T or later
- with interworking support, that is /interwork, on ARMv5T and later. ARMv5T and later architectures provide direct support to interworking by using instructions such as BLX and load to program counter instructions.

/ropi, /noropi

If you select the /ropi qualifier to generate ROPI code, the compiler:

- addresses read-only code and data PC-relative
- sets the *Position Independent* (PI) attribute on read-only output sections.

———— **Note** —————

--apcs=/ropi is not supported when compiling C++.

/rwp, /norwp

If you select the /rwp qualifier to generate RWPI code, the compiler:

- addresses writable data using offsets from the static base register sb. This means that:
 - the base address of the RW data region can be fixed at runtime
 - data can have multiple instances
 - data can be, but does not have to be, position-independent.
- sets the PI attribute on read/write output sections.

———— **Note** —————

Because the --lower_rwp option is the default, code that is not RWPI is automatically transformed into equivalent code that is RWPI. This static initialization is done at runtime by the C++ constructor mechanism, even for C.

/fpic, /nofpic

If you select this option, the compiler:

- accesses all static data using PC-relative addressing
- accesses all imported or exported read-write data using a *Global Offset Table* (GOT) entry created by the linker
- accesses all read-only data relative to the PC.

You must compile your code with /fpic if it uses shared objects. This is because relative addressing is only implemented when your code makes use of System V shared libraries.

You do not have to compile with /fpic if you are building either a static image or static library.

The use of /fpic is supported when compiling C++. In this case, virtual function tables and typeinfo are placed in read-write areas so that they can be accessed relative to the location of the PC.

Note

When building a System V or ARM Linux shared library, use `--apcs /fpic` together with `--no_hide_all`.

`/hardfp`

If you use `/hardfp`, the compiler generates code for hardware floating-point linkage. Hardware floating-point linkage uses the FPU registers to pass the arguments and return values.

`/hardfp` interacts with or overrides explicit or implicit use of `--fpu` as follows:

- If floating-point support is not permitted (for example, because `--fpu=none` is specified, or because of other means), `/hardfp` is ignored.
- If floating-point support is permitted, but without floating-point hardware (`--fpu=softvfp`), `/hardfp` gives an error.
- If floating-point hardware is available and the *hardfp* calling convention is used (`--fpu=vfp...`), `/hardfp` is ignored.
- If floating-point hardware is present and the *softfp* calling convention is used (`--fpu=softvfp+vfp...`), `/hardfp` gives an error.

The `/hardfp` and `/softfp` qualifiers are mutually exclusive.

`/softfp`

If you use `/softfp`, software floating-point linkage is used. Software floating-point linkage means that the parameters and return value for a function are passed using the ARM integer registers `r0` to `r3` and the stack.

`/softfp` interacts with or overrides explicit or implicit use of `--fpu` as follows:

- If floating-point support is not permitted (for example, because `--fpu=none` is specified, or because of other means), `/softfp` is ignored.
- If floating-point support is permitted, but without floating-point hardware (`--fpu=softvfp`), `/softfp` is ignored because the state is already `/softfp`.
- If floating-point hardware is present, `/softfp` forces the *softfp* (`--fpu=softvfp+vfp...`) calling convention.

The `/hardfp` and `/softfp` qualifiers are mutually exclusive.

3.6.4 Restrictions

There are restrictions when you compile code with `/ropi`, or `/rwp_i`, or `/fpic`.

`/ropi`

The main restrictions when compiling with `/ropi` are:

- The use of `--apcs=/ropi` is not supported when compiling C++. You can compile only the C subset of C++ with `/ropi`.
- Some constructs that are legal C do not work when compiled for `--apcs=/ropi`. For example:

```
extern const int ci; // ro
const int *p2 = &ci; // this static initialization
                    // does not work with --apcs=/ropi
```

To enable such static initializations to work, compile your code using the `--lower_ropi` option. For example:

```
armcc --apcs=/ropi --lower_ropi
```

`/rwp`The main restrictions when compiling with `/rwp` are:

- Some constructs that are legal C do not work when compiled for `--apcs=/rwp`. For example:

```
int i;           // rw
int *p1 = &i;   // this static initialization
                // does not work with --apcs=/rwp
                // --no_lower_rwp
```

To enable such static initializations to work, compile your code using the `--lower_rwp` option. For example:

```
armcc --apcs=/rwp
```

———— **Note** —————

You do not have to specify `--lower_rwp`, because this is the default.

`/fpic`The main restrictions when compiling with `/fpic` are:

- By default, if you use `--apcs=/fpic`, the compiler exports only functions and data marked `__declspec(dllexport)`.
- If you use `--apcs=/fpic` and `--no_hide_all` on the same command line, the compiler uses default ELF dynamic visibility for all extern variables and functions that do not use `__declspec(dllexport)`. The compiler disables auto-inlining for functions with default ELF visibility.

3.6.5 See also

- [--fpu=name](#) on page 3-100
- [--hide_all, --no_hide_all](#) on page 3-113
- [--lower_ropi, --no_lower_ropi](#) on page 3-141
- [--lower_rwp, --no_lower_rwp](#) on page 3-142
- [__declspec\(dllexport\)](#) on page 5-31
- [Compiler options for floating-point linkage and computations](#) on page 6-67
- [Default selection of hardware or software floating-point support](#) on page 6-57 in *Using the Compiler*
- [ARM C libraries and multithreading](#) on page 2-16 in *Using ARM C and C++ Libraries and Floating-Point Support*
- [Overview of veneers](#) on page 4-26 in *Using the Linker*
- [Chapter 10 BPABI and SysV shared libraries and executables](#) in *Using the Linker*
- [Procedure Call Standard for the ARM architecture](#) in `install_directory\Documentation\Specifications\...`

3.7 --arm

This option is a request to the compiler to target the ARM instruction set. The compiler is permitted to generate both ARM and Thumb code, but recognizes that ARM code is preferred.

———— **Note** —————

This option is not relevant for Thumb-only processors such as Cortex-M4, Cortex-M3, Cortex-M1, and Cortex-M0.

3.7.1 Default

This is the default option for targets supporting the ARM instruction set.

3.7.2 See also

- [--arm_only](#) on page 3-23
- [--cpu=list](#) on page 3-48
- [--cpu=name](#) on page 3-49
- [--thumb](#) on page 3-197
- [#pragma arm](#) on page 5-87
- [ARM architectures supported by the toolchain](#) on page 2-17 in *Introducing the ARM Compiler toolchain*.

3.8 --arm_linux

This option configures a set of other options with defaults that are suitable for ARM Linux compilation.

3.8.1 Usage

These defaults are enabled automatically when you use one of the following ARM Linux options:

- `--arm_linux_paths`
- `--translate_gcc` in full GCC emulation mode
- `--translate_g++` in full GCC emulation mode
- `--translate_gld` in full GCC emulation mode.

Typical use of this option is to aid the migration of legacy code. It enables you to simplify the compiler options used in existing makefiles, while retaining full and explicit control over the header and library search paths used.

When migrating from a build earlier than RVCT v4.0, you can replace all of these options supplied to the compiler with a single `--arm_linux` option.

3.8.2 Default

By default, the configured set of options is:

- `--apcs=/interwork`
- `--enum_is_int`
- `--gnu`
- `--library_interface=aeabi_glibc`
- `--no_execstack`
- `--no_hide_all`
- `--preinclude=linux_armcc.h`
- `--wchar32.`

3.8.3 Example

To apply the default set of options, use `--arm_linux`.

To override any of the default options, specify them separately. For example, `--arm_linux --hide_all`.

In the latter example, `--hide_all` overrides the `--no_hide_all` encompassed by `--arm_linux`.

3.8.4 See also

- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--configure_gcc=path](#) on page 3-43
- [--configure_gcc_version=version](#) on page 3-44
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46

- [--execstack, --no_execstack](#) on page 3-89
- [--gnu_defaults](#) on page 3-108
- [--shared](#) on page 3-188
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries, --no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.9 --arm_linux_config_file=path

This option specifies the location of the configuration file that is created for ARM Linux builds. It enables the use of standard Linux configuration settings when compiling your code.

3.9.1 Syntax

```
--arm_linux_config_file=path
```

Where *path* is the path and filename of the configuration file.

3.9.2 Restrictions

You must use this option both when generating the configuration file and when using the configuration during compilation and linkage.

If you specify an ARM Linux configuration file on the command line and you use `--translate_gcc`, `--translate_g++`, or `--translate_gld`, you affect the default settings for certain other options. The default value for `--bss_threshold` becomes zero, the default for `--signed_bitfields` and `--unsigned_bitfields` becomes `--signed_bitfields`, and `--enum_is_int` and `--wchar32` are switched on.

3.9.3 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--bss_threshold=num](#) on page 3-30
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--configure_gcc=path](#) on page 3-43
- [--configure_gcc_version=version](#) on page 3-44
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--enum_is_int](#) on page 3-85
- [--gnu_defaults](#) on page 3-108
- [--shared](#) on page 3-188
- [--signed_bitfields](#), [--unsigned_bitfields](#) on page 3-190
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [--wchar32](#) on page 3-225
- [-Warmcc](#), [--gcc_fallback](#) on page 3-222
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries](#), [--no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.10 --arm_linux_configure

This option configures the tools for use with ARM Linux by creating a configuration file describing include paths, library paths, and standard libraries for the GNU C library, `glibc`. The created configuration file is used when you build your code.

3.10.1 Usage

Automatic and manual methods of configuration apply. Automatic configuration attempts to automatically locate an installation of the GNU toolchain on your `PATH` environment variable, and query it to determine the configuration settings to use. Manual configuration can be used to specify your own locations for header files and libraries. It can be used if you do not have a complete GNU toolchain installed.

If you use automatic configuration, the GCC version number of the GNU toolchain is added to the configuration file. The corresponding `--gnu_version=version` option is passed to the compiler from the configuration file when using any of the translation options or `--arm_linux_paths`.

To perform automatic configuration:

- `armcc --arm_linux_configure --arm_linux_config_file=config_file_path --configure_gcc=path --configure_gld=path`

where *config_file_path* is the path and filename of the configuration file that is created. You can optionally specify the location of the *GNU Compiler Collection* (GCC) driver, and optionally the location of the GNU linker, to override the locations determined from the system `PATH` environment variable.

To perform manual configuration:

- `armcc --arm_linux_configure --arm_linux_config_file=path --configure_cpp_headers=path --configure_sysroot=path`

where the paths to the GNU `libstdc++ Standard Template Library` (STL) header files, and the system root path that libraries and header files are found from, are specified.

3.10.2 Restrictions

A GNU toolchain must exist on your system to use automatic configuration.

If using the automatic method of configuration, an ARM Linux GCC must be located with the system `PATH` environment variable. If you do not have a suitable GCC on your system path, you can either add one to your path, or use `--configure_gcc` (and optionally `--configure_gld`) to manually specify the location of a suitable GCC.

3.10.3 Default

Automatic configuration applies unless you specify the location of GCC or the GNU linker using additional options. That is, the compiler attempts to locate an ARM Linux GCC using your system path environment variable, unless you use additional options to specify otherwise.

3.10.4 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_paths](#) on page 3-21
- [--configure_gcc=path](#) on page 3-43
- [--configure_gcc_version=version](#) on page 3-44

- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--gnu_defaults](#) on page 3-108
- [--gnu_version=version](#) on page 3-110
- [--shared](#) on page 3-188
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [-Warmcc,--gcc_fallback](#) on page 3-222
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries, --no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.11 --arm_linux_paths

This option enables you to build code for ARM Linux.

3.11.1 Usage

You can use this option after you have configured the tools for use with ARM Linux.

This is a compiler option only. It follows the typical GCC usage model, where the compiler driver is used to direct linkage and selection of standard system object files and libraries.

This option can also be used to aid migration from versions of RVCT earlier than RVCT v4.0. After you have created a configuration file using `--arm_linux_configure`, you can modify an existing build by replacing the list of standard options and search paths with the `--arm_linux_paths` option. That is, `--arm_linux_paths` can be used to replace:

- all of the default options listed for `--arm_linux`
- header paths
- library paths
- standard libraries.

3.11.2 Restrictions

You must specify the location of the configuration file by using `--arm_linux_config_file=filename`.

3.11.3 Examples

Compile and link application code:

```
armcc --arm_linux_paths --arm_linux_config_file=my_config_file -o hello -O2 -Otime -g
hello.c
```

Compile a source file `source.c` for use in a shared library:

```
armcc --arm_linux_paths --arm_linux_config_file=my_config_file --apcs=/fpic -c source.c
```

Link two object files, `obj1` and `obj2`, into a shared library called `my_shared_lib.so`, using the compiler:

```
armcc --arm_linux_paths --arm_linux_config_file=my_config_file --shared -o
my_shared_lib.so obj1.o obj2.o
```

3.11.4 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--configure_gcc=path](#) on page 3-43
- [--configure_gcc_version=version](#) on page 3-44
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--gnu_defaults](#) on page 3-108
- [--shared](#) on page 3-188

- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [-Warmcc,--gcc_fallback](#) on page 3-222
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries, --no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.12 --arm_only

This option enforces ARM-only code. The compiler behaves as if Thumb is absent from the target architecture.

The compiler propagates the --arm_only option to the assembler and the linker.

3.12.1 Default

For targets that support the ARM instruction set, the default is --arm. For targets that do not support the ARM instruction set, the default is --thumb.

3.12.2 Example

```
armcc --arm_only myprog.c
```

———— **Note** —————

If you specify `armcc --arm_only --thumb myprog.c`, this does *not* mean that the compiler checks your code to ensure that no Thumb code is present. It means that --thumb overrides --arm_only, because of command-line ordering.

3.12.3 See also

- [--arm on page 3-15](#)
- [--thumb on page 3-197](#)
- [Assembler command line options on page 2-3](#) in the *Assembler Reference* for information on --16 and --32
- [About ordering the compilation tools command-line options on page 2-22](#) in *Introducing ARM Compilation Tools*.

3.13 --asm

This option instructs the compiler to write a listing to a file of the disassembly of the machine code generated by the compiler.

Object code is generated when this option is selected. The link step is also performed, unless the `-c` option is chosen.

Note

To produce a disassembly of the machine code generated by the compiler, without generating object code, select `-S` instead of `--asm`.

3.13.1 Usage

The action of `--asm`, and the full name of the disassembly file produced, depends on the combination of options used:

Table 3-1 Compiling with the `--asm` option

Compiler option	Action
<code>--asm</code>	Writes a listing to a file of the disassembly of the compiled source. The link step is also performed, unless the <code>-c</code> option is used. The disassembly is written to a text file whose name defaults to the name of the input file with the filename extension <code>.s</code> .
<code>--asm -c</code>	As for <code>--asm</code> , except that the link step is not performed.
<code>--asm --interleave</code>	As for <code>--asm</code> , except that the source code is interleaved with the disassembly. The disassembly is written to a text file whose name defaults to the name of the input file with the filename extension <code>.txt</code> .
<code>--asm --multifile</code>	As for <code>--asm</code> , except that the compiler produces empty object files for the files merged into the main file.
<code>--asm -o filename</code>	As for <code>--asm</code> , except that the object file is named <i>filename</i> . The disassembly is written to the file <i>filename.s</i> . The name of the object file must not have the filename extension <code>.s</code> . If the filename extension of the object file is <code>.s</code> , the disassembly is written over the top of the object file. This might lead to UNPREDICTABLE results.

3.13.2 See also

- [-c on page 3-31](#)
- [--interleave on page 3-124](#)
- [--multifile, --no_multifile on page 3-150](#)
- [-o filename on page 3-154](#)
- [-S on page 3-187](#)
- [Filename suffixes recognized by the compiler on page 3-15](#) in *Using the Compiler*.

3.14 `--asm_dir=directory_name`

This option enables you to specify a directory for output assembler files.

3.14.1 Example

```
armcc -c --output_dir=obj -S f1.c f2.c --asm_dir=asm
```

Result:

```
asm/f1.s  
asm/f2.s  
obj/f1.o  
obj/f2.o
```

3.14.2 See also

- [--depend_dir=directory_name](#) on page 3-62
- [--list_dir=directory_name](#) on page 3-135
- [--output_dir=directory_name](#) on page 3-162.

3.15 `--autoinline`, `--no_autoinline`

These options enable and disable automatic inlining of functions.

The compiler automatically inlines functions at the higher optimization levels where it is sensible to do so. The `-Ospace` and `-Otime` options, together with some other factors such as function size, influence how the compiler automatically inlines functions.

Selecting `-Otime`, in combination with various other factors, increases the likelihood that functions are inlined.

In general, when automatic inlining is enabled, the compiler inlines any function that is sensible to inline. When automatic inlining is disabled, only functions marked as `__inline` are candidates for inlining.

3.15.1 Usage

Use these options to control the automatic inlining of functions at the highest optimization levels (`-O2` and `-O3`).

3.15.2 Default

For optimization levels `-O0` and `-O1`, the default is `--no_autoinline`.

For optimization levels `-O2` and `-O3`, the default is `--autoinline`.

3.15.3 See also

- [--forceinline](#) on page 3-95
- [--inline](#), [--no_inline](#) on page 3-122
- [-Onum](#) on page 3-156
- [-Ospace](#) on page 3-160
- [-Otime](#) on page 3-161
- [Default compiler options that are affected by optimization level on page 5-45](#) in *Using the Compiler*.

3.16 `--bigend`

This option instructs the compiler to generate code for an ARM processor using big-endian memory.

The ARM architecture defines the following big-endian modes:

- BE8** Byte Invariant Addressing mode (ARMv6 and later).
- BE32** Legacy big-endian mode.

The selection of BE8 versus BE32 is specified at link time.

3.16.1 Default

The compiler assumes `--littleend` unless `--bigend` is explicitly specified.

3.16.2 See also

- [--littleend on page 3-137](#)
- [ARM architecture v4T on page 2-11](#) in *Developing Software for ARM Processors*
- [--be8 on page 2-19](#) in the *Linker Reference*
- [--be32 on page 2-20](#) in the *Linker Reference*.

3.17 --bitband

This option bit-bands all non **const** global structure objects. It enables a word of memory to be mapped to a single bit in the bit-band region. This enables efficient atomic access to single-bit values in SRAM and Peripheral regions of the memory architecture.

For peripherals that are width sensitive, byte, halfword, and word stores or loads to the alias space are generated for **char**, **short**, and **int** types of bitfields of bit-banded structs respectively.

3.17.1 Restrictions

The following restrictions apply:

- This option only affects **struct** types. Any union type or other aggregate type with a union as a member cannot be bit-banded.
- Members of structs cannot be bit-banded individually.
- Bit-banded accesses are generated only for single-bit bitfields.
- Bit-banded accesses are not generated for **const** objects, pointers, and local objects.
- Bit-banding is only available on some processors. For example, the Cortex-M4 and Cortex-M3 processors.

3.17.2 Example

In [Example 3-1](#) the writes to bitfields *i* and *k* are bit-banded when compiled using the `--bitband` command-line option.

Example 3-1 Bit-banding example

```
typedef struct {
    int i : 1;
    int j : 2;
    int k : 1;
} BB;

BB value;

void update_value(void)
{
    value.i = 1;
    value.k = 1;
}
```

3.17.3 See also

- [__attribute__\(\(bitband\)\)](#) type attribute on page 5-65
- [Compiler and processor support for bit-banding on page 5-24](#) in *Using the Compiler*
- the *Technical Reference Manual* for your processor.

3.18 --brief_diagnostics, --no_brief_diagnostics

This option enables or disables the output of brief diagnostic messages by the compiler.

When enabled, the original source line is not displayed, and error message text is not wrapped if it is too long to fit on a single line.

3.18.1 Default

The default is `--no_brief_diagnostics`.

3.18.2 Example

```
/* main.c */
#include <stdio.h>
int main(void)
{
    printf("Hello, world\n"); // Intentional quotation mark error
    return 0;
}
```

Compiling this code with `--brief_diagnostics` produces:

```
"main.c", line 5: Error: #18: expected a ")"
"main.c", line 5: Error: #7: unrecognized token
"main.c", line 5: Error: #8: missing closing quote
"main.c", line 6: Error: #65: expected a ";"
```

3.18.3 See also

- [--diag_error=tag\[,tag,...\]](#) on page 3-70
- [--diag_remark=tag\[,tag,...\]](#) on page 3-71
- [--diag_style={arm|ide|gnu}](#) on page 3-72
- [--diag_suppress=tag\[,tag,...\]](#) on page 3-73
- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [--errors=filename](#) on page 3-86
- [--remarks](#) on page 3-181
- [-W](#) on page 3-220
- [--wrap_diagnostics, --no_wrap_diagnostics](#) on page 3-228
- [Chapter 7 Compiler Diagnostic Messages](#) in *Using the Compiler*.

3.19 --bss_threshold=num

This option controls the placement of small global ZI data items in sections. A *small global ZI data item* is an uninitialized data item that is eight bytes or less in size.

3.19.1 Syntax

```
--bss_threshold=num
```

Where:

<i>num</i>	is either:
0	place small global ZI data items in ZI data sections
8	place small global ZI data items in RW data sections.

3.19.2 Usage

In ARM Compiler 4.1 and later, the compiler might place small global ZI data items in RW data sections as an optimization. In RVCT 2.0.1 and earlier, small global ZI data items were placed in ZI data sections by default.

Use `--bss_threshold=0` to emulate the behavior of RVCT 2.0.1 and earlier with respect to the placement of small global ZI data items in ZI data sections.

———— **Note** —————

Selecting the option `--bss_threshold=0` instructs the compiler to place *all* small global ZI data items in the current compilation module in a ZI data section. To place specific variables in:

- a ZI data section, use `__attribute__((zero_init))`
- a specific ZI data section, use a combination of `__attribute__((section("name")))` and `__attribute__((zero_init))`.

3.19.3 Default

If you do not specify a `--bss_threshold` option, the compiler assumes `--bss_threshold=8`.

If you specify an ARM Linux configuration file on the command line and you use `--translate_gcc` or `--translate_g++`, the compiler assumes `--bss_threshold=0`.

3.19.4 Example

```
int glob1;          /* ZI (.bss) in RVCT 2.0.1 and earlier */
                   /* RW (.data) in RVCT 2.1 and later */
```

Compiling this code with `--bss_threshold=0` places `glob1` in a ZI data section.

3.19.5 See also

- [#pragma arm section \[section_type_list\]](#) on page 5-88
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [__attribute__\(\(section\("name"\)\)\) variable attribute](#) on page 5-77
- [__attribute__\(\(zero_init\)\) variable attribute](#) on page 5-84.

3.20 -c

This option instructs the compiler to perform the compilation step, but not the link step.

———— **Note** —————

This option is different from the uppercase -C option.

3.20.1 Usage

The use of the -c option is recommended in projects with more than one source file.

3.20.2 See also

- [--asm on page 3-24](#)
- [--list on page 3-133](#)
- [-o filename on page 3-154](#)
- [-S on page 3-187.](#)

3.21 -C

This option instructs the compiler to retain comments in preprocessor output.

Choosing this option implicitly selects the option -E.

———— **Note** —————

This option is different from the lowercase -c option.

3.21.1 See also

- [-E on page 3-82.](#)

3.22 --c90

This option enables the compilation of C90 source code. It enforces C only, and C++ syntax is not accepted.

3.22.1 Usage

This option can also be combined with other source language command-line options. For example, `armcc --c90 --gnu`.

To ensure conformance with ISO/IEC 9899:1990, the 1990 International Standard for C and ISO/IEC 9899 AM1, the 1995 Normative Addendum 1, you must also use the `--strict` option.

3.22.2 Default

This option is implicitly selected for files having a suffix of `.c`, `.ac`, or `.tc`.

Note

If you are migrating from RVCT, be aware that filename extensions `.ac` and `.tc` are deprecated in ARM Compiler 4.1 and later.

3.22.3 See also

- [--c99 on page 3-34](#)
- [--gnu on page 3-107](#)
- [--strict, --no_strict on page 3-194](#)
- [Source language modes of the compiler on page 2-3](#)
- [Filename suffixes recognized by the compiler on page 3-15](#) in *Using the Compiler*.

3.23 --c99

This option enables the compilation of C99 source code. It enforces C only, and C++ syntax is not accepted.

3.23.1 Usage

This option can also be combined with other source language command-line options. For example, `armcc --c99 --gnu`.

To ensure conformance with the ISO/IEC 9899:1999, the 1999 International Standard for C, you must also use the `--strict` option.

3.23.2 Default

For files having a suffix of `.c`, `.ac`, or `.tc`, `--c99` applies by default.

3.23.3 See also

- [--c90 on page 3-33](#)
- [--gnu on page 3-107](#)
- [--strict, --no_strict on page 3-194](#)
- [Source language modes of the compiler on page 2-3](#).

3.24 --code_gen, --no_code_gen

This option enables or disables the generation of object code.

When generation of object code is disabled, the compiler performs syntax-checking only, without creating an object file.

3.24.1 Default

The default is --code_gen.

3.25 --compatible=*name*

This option generates code that is compatible with multiple target architectures or processors.

3.25.1 Syntax

`--compatible=name`

Where:

name is the name of a target processor or architecture, or NONE. Processor and architecture names are not case-sensitive.

If multiple instances of this option are present on the command line, the last one specified overrides the previous instances.

Specify `--compatible=NONE` at the end of the command line to turn off all other instances of the option.

3.25.2 Usage

Using this option avoids the need to recompile the same source code for different targets. You could apply this use to a possible target upgrade where a different architecture or processor is to be used in the future, without having to separately recompile for that target.

See [Table 3-2](#). The valid combinations are:

- `--cpu=CPU_from_group1 --compatible=CPU_from_group2`
- `--cpu=CPU_from_group2 --compatible=CPU_from_group1`.

Table 3-2 Compatible processor or architecture combinations

Group 1	ARM7TDMI, 4T
Group 2	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4, 7-M, 6-M, 6S-M, SC300, SC000

No other combinations are permitted.

The effect is to compile code that is compatible with both `--cpu` and `--compatible`. This means that only 16-bit Thumb instructions are used. (This is the intersection of the capabilities of group 1 and group 2.)

———— **Note** —————

Although the generated code is compatible with multiple targets, this code might be less efficient than compiling for a single target processor or architecture.

3.25.3 Example

This example gives code that is compatible with both the ARM7TDMI processor and the Cortex-M4 processor.

```
armcc --cpu=arm7tdmi --compatible=cortex-m4 myprog.c
```

3.25.4 See also

- [--cpu=*name*](#) on page 3-49.

3.26 `--compile_all_input`, `--no_compile_all_input`

These options enable and disable the suppression of filename extension processing, enabling the compiler to compile files with any filename extensions.

When enabled, the compiler suppresses filename extension processing entirely, treating all input files as if they have the suffix `.c`.

3.26.1 Default

The default is `--no_compile_all_input`.

3.26.2 See also

- [--link_all_input](#), [--no_link_all_input](#) on page 3-132
- [Filename suffixes recognized by the compiler](#) on page 3-15 in *Using the Compiler*.

3.27 --conditionalize, --no_conditionalize

These options enable and disable the generation of conditional instructions, that is instructions with the condition code suffix.

--conditionalize enables the compiler to generate conditional instructions such as ADDEQ and LDRGE.

When you compile with --no_conditionalize, the compiler does not generate conditional instructions such as ADDEQ and LDRGE. It generates conditional branch instructions such as BEQ and BLGE to execute conditional code. The only instructions that can be conditional are B, BL, BX, BLX, and BXJ.

3.27.1 Default

The default is --conditionalize.

3.27.2 See also

- [Conditional instructions on page 6-2](#) in *Using the Assembler*
- [Condition code suffixes on page 6-6](#) in *Using the Assembler*
- [Condition code meanings on page 6-8](#) in *Using the Assembler*.

3.28 `--configure_cpp_headers=path`

This option specifies the path to the GNU libstdc++ STL header files, when configuring the tools for use with ARM Linux.

3.28.1 Syntax

```
--configure_cpp_headers=path
```

Where:

`path` is the path to the GNU C++ STL header files.

3.28.2 Usage

This option overrides any path that is automatically detected. It can be used as part of a manual approach to configuring the tools for use with ARM Linux.

3.28.3 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--configure_gcc=path](#) on page 3-43
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--gnu_defaults](#) on page 3-108
- [--shared](#) on page 3-188
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [--arm_linux](#) on page 2-13 in the *Linker Reference Guide*
- [--library=name](#) on page 2-97 in the *Linker Reference Guide*
- [--search_dynamic_libraries, --no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference Guide*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.29 `--configure_extra_includes=paths`

This option specifies any additional system include paths when configuring the tools for use with ARM Linux.

3.29.1 Syntax

`--configure_extra_includes=paths`

Where:

`paths` is a comma separated list of pathnames denoting the locations of the additional system include paths.

3.29.2 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_libraries=paths](#) on page 3-41
- [--configure_gcc=path](#) on page 3-43
- [--configure_gcc_version=version](#) on page 3-44
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--gnu_defaults](#) on page 3-108
- [--shared](#) on page 3-188
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries](#), [--no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.30 `--configure_extra_libraries=paths`

This option specifies any additional system library paths when configuring the tools for use with ARM Linux.

3.30.1 Syntax

`--configure_extra_libraries=paths`

Where:

`paths` is a comma separated list of pathnames denoting the locations of the additional system library paths.

3.30.2 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_gcc=path](#) on page 3-43
- [--configure_gcc_version=version](#) on page 3-44
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--gnu_defaults](#) on page 3-108
- [--shared](#) on page 3-188
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries](#), [--no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.31 `--configure_gas=path`

This option specifies the location of the *GNU assembler* (*gas*), when configuring the tools for use with ARM Linux.

3.31.1 Usage

To optionally invoke *gas* rather than *armasm* when compiling source files ending in *.s* or *.S*, you can either:

- specify `--configure_gas=path` when using `--arm_linux_configure`
- rely on the Linux configuration to query GCC for the path to the *gas* executable.

Specifying `--configure_gas=path` overrides the Linux configuration querying GCC for the path to the *gas* executable.

During translation, invoke *gas* by using `-Warmcc,--use_gas`.

3.31.2 See also

- [--use_gas](#) on page 3-210
- [-Warmcc,option\[,option,...\]](#) on page 3-221.

3.32 `--configure_gcc=path`

This option specifies the location of the GCC driver, when configuring the tools for use with ARM Linux.

3.32.1 Syntax

`--configure_gcc=path`

Where:

`path` is the path and filename of the GCC driver.

3.32.2 Usage

Use this option if you want to override the default location of the GCC driver specified during configuration, or if the automatic configuration method of `--arm_linux_configure` fails to find the driver.

3.32.3 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--configure_gcc_version=version](#) on page 3-44
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--gnu_defaults](#) on page 3-108
- [--shared](#) on page 3-188
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [-Warmcc,--gcc_fallback](#) on page 3-222
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries, --no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.33 `--configure_gcc_version=version`

This option enables you to manually set, or override, the GCC version when configuring for ARM Linux.

If you use this option to override the reported version when configuring against a GCC installation, the compiler gives a warning if the override version you specify is older than the version of the GCC installation.

3.33.1 Syntax

`--configure_gcc_version=version`

Where:

`version` is a GCC version number of the form $N.[M]N.[M]N$.

3.33.2 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--configure_gcc=path](#) on page 3-43
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--gnu_defaults](#) on page 3-108
- [--shared](#) on page 3-188
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries, --no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.34 `--configure_gld=path`

This option specifies the location of the GNU linker, `ld`.

3.34.1 Syntax

`--configure_gld=path`

Where:

`path` is the path and filename of the GNU linker.

3.34.2 Usage

During configuration, the compiler attempts to determine the location of the GNU linker used by GCC. If the compiler is unable to determine the location, or if you want to override the normal path to the GNU linker, you can specify its location by using the `--configure_gld=path` option. The path is the full path and filename of the GNU `ld` binary.

3.34.3 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--configure_gcc=path](#) on page 3-43
- [--configure_gcc_version=version](#) on page 3-44
- [--configure_sysroot=path](#) on page 3-46
- [--gnu_defaults](#) on page 3-108
- [--shared](#) on page 3-188
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries](#), [--no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.35 --configure_sysroot=*path*

This option specifies the system root path to use when configuring the tools for use with ARM Linux.

3.35.1 Syntax

```
--configure_sysroot=path
```

Where *path* is the system root path to use.

3.35.2 Usage

This option overrides any system root path that is automatically detected. It can be used as part of a manual approach to configuring the tools for use with ARM Linux if you want to use a different path to your normal system root path.

The system root path is the base path that libraries and header files are normally found from. On a standard Linux system, this is typically the root of the file system. In a cross compilation GNU toolchain, it is usually the parent directory of the GNU C library installation. This directory contains the `lib`, `usr/lib`, and `usr/include` subdirectories that hold the C libraries and header files.

3.35.3 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=*path*](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--configure_cpp_headers=*path*](#) on page 3-39
- [--configure_extra_includes=*paths*](#) on page 3-40
- [--configure_extra_libraries=*paths*](#) on page 3-41
- [--configure_gcc=*path*](#) on page 3-43
- [--configure_gcc_version=*version*](#) on page 3-44
- [--configure_gld=*path*](#) on page 3-45
- [--gnu_defaults](#) on page 3-108
- [--shared](#) on page 3-188
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=*name*](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries](#), [--no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.36 --cpp

This option enables the compilation of C++ source code.

3.36.1 Usage

This option can also be combined with other source language command-line options. For example, `armcc --cpp --gnu`.

3.36.2 Default

This option is implicitly selected for files having a suffix of `.cpp`, `.cxx`, `.c++`, `.cc`, or `.CC`.

3.36.3 See also

- [--anachronisms, --no_anachronisms](#) on page 3-10
- [--c90](#) on page 3-33
- [--c99](#) on page 3-34
- [--gnu](#) on page 3-107
- [--strict, --no_strict](#) on page 3-194
- [Source language modes of the compiler](#) on page 2-3.

3.37 --cpu=list

This option lists the supported architecture and processor names that can be used with the `--cpu=name` option.

3.37.1 See also

- [--cpu=name](#) on page 3-49
- [Processors and their implicit Floating-Point Units \(FPUs\) on page 6-71](#) in *Using the Compiler*.

3.38 --cpu=*name*

This option enables code generation for the selected ARM processor or architecture.

3.38.1 Syntax

--cpu=*name*

Where:

name is the name of a processor or architecture.

If *name* is the name of a processor, enter it as shown on ARM data sheets, for example, ARM7TDMI, ARM1176JZ-S, MPCore.

If *name* is the name of an architecture, it must belong to the list of architectures shown in [Table 3-3](#).

Processor and architecture names are not case-sensitive.

Wildcard characters are not accepted.

Table 3-3 Supported ARM architectures

Architecture	Description	Example processors
4	ARMv4 without Thumb	SA-1100
4T	ARMv4 with Thumb	ARM7TDMI, ARM9TDMI, ARM720T, ARM740T, ARM920T, ARM922T, ARM940T, SC100
5T	ARMv5 with Thumb and interworking	-
5TE	ARMv5 with Thumb, interworking, DSP multiply, and double-word instructions	ARM9E, ARM946E-S, ARM966E-S
5TEJ	ARMv5 with Thumb, interworking, DSP multiply, double-word instructions, and Jazelle [®] extensions ^a	ARM926EJ-S, ARM1026EJ-S, SC200
6	ARMv6 with Thumb, interworking, DSP multiply, double-word instructions, unaligned and mixed-endian support, Jazelle, and media extensions	ARM1136J-S, ARM1136JF-S
6-M	ARMv6 micro-controller profile with Thumb only, plus processor state instructions	Cortex-M1 without OS extensions, Cortex-M0, SC000, Cortex-M0plus
6S-M	ARMv6 micro-controller profile with Thumb only, plus processor state instructions and OS extensions	Cortex-M1 with OS extensions
6K	ARMv6 with SMP extensions	MPCore
6T2	ARMv6 with Thumb (Thumb-2 technology)	ARM1156T2-S, ARM1156T2F-S
6Z	ARMv6 with Security Extensions	ARM1176JZF-S, ARM1176JZ-S
7	ARMv7 with Thumb (Thumb-2 technology) only, and without hardware divide	-
7-A	ARMv7 application profile supporting virtual MMU-based memory systems, with ARM, Thumb (Thumb-2 technology) and ThumbEE, DSP support, and 32-bit SIMD support	Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A15

Table 3-3 Supported ARM architectures (continued)

Architecture	Description	Example processors
7-A.security	Enables the use of the SMC instruction (formerly SMI) when assembling for the v7-A architecture	Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A15
7-R	ARMv7 real-time profile with ARM, Thumb (Thumb-2 technology), DSP support, and 32-bit SIMD support	Cortex-R4, Cortex-R4F, Cortex-R7
7-M	ARMv7 micro-controller profile with Thumb (Thumb-2 technology) only and hardware divide	Cortex-M3, SC300
7E-M	ARMv7-M enhanced with DSP (saturating and 32-bit SIMD) instructions	Cortex-M4

a. The ARM compiler cannot generate Java bytecodes.

Note

- ARMv7 is not an actual ARM architecture. `--cpu=7` denotes the features that are common to the ARMv7-A, ARMv7-R, and ARMv7-M architectures. By definition, any given feature used with `--cpu=7` exists on the ARMv7-A, ARMv7-R, and ARMv7-M architectures.
 - 7-A.security is not an actual ARM architecture, but rather, refers to 7-A plus Security Extensions.
-

3.38.2 Default

If you do not specify a `--cpu` option, the compiler assumes `--cpu=ARM7TDMI`.

To obtain a full list of CPU architectures and processors, use the `--cpu=list` option.

3.38.3 Usage

The following general points apply to processor and architecture options:

Processors

- Selecting the processor selects the appropriate architecture, *Floating-Point Unit* (FPU), and memory organization.
- The supported `--cpu` values include all current ARM product names or architecture versions.
Other ARM architecture-based processors, such as the Marvell Feroceon and the Marvell XScale, are also supported.
- If you specify a processor for the `--cpu` option, the compiled code is optimized for that processor. This enables the compiler to use specific coprocessors or instruction scheduling for optimum performance.

Architectures

- If you specify an architecture name for the `--cpu` option, the code is compiled to run on any processor supporting that architecture. For example, `--cpu=5TE` produces code that can be used by the ARM926EJ-S®.

FPU

- Some specifications of `--cpu` imply an `--fpu` selection. For example, when compiling with the `--arm` option, `--cpu=ARM1136JF-S` implies `--fpu=vfpv2`. Similarly, `--cpu=Cortex-R4F` implies `--fpu=vfpv3_d16`.

Note

Any *explicit* FPU, set with `--fpu` on the command line, overrides an *implicit* FPU.

- If no `--fpu` option is specified and no `--cpu` option is specified, `--fpu=softvfp` is used.

ARM/Thumb

- Specifying a processor or architecture that supports Thumb instructions, such as `--cpu=ARM7TDMI`, does not make the compiler generate Thumb code. It only enables features of the processor to be used, such as long multiply. Use the `--thumb` option to generate Thumb code, unless the processor is a Thumb-only processor, for example Cortex-M4. In this case, `--thumb` is not required.

Note

Specifying the target processor or architecture might make the object code generated by the compiler incompatible with other ARM processors. For example, code compiled for architecture ARMv6 might not run on an ARM920T processor, if the compiled code includes instructions specific to ARMv6. Therefore, you must choose the lowest common denominator processor suited to your purpose.

- If you are compiling code that is intended for mixed ARM/Thumb systems for processors that support ARMv4T or ARMv5T, then you must specify the interworking option `--apcs=/interwork`. By default, this is enabled for processors that support ARMv5T or above.
- If you compile for Thumb, that is with the `--thumb` option on the command line, the compiler compiles as much of the code as possible using the Thumb instruction set. However, the compiler might generate ARM code for some parts of the compilation. For example, if you are compiling code for a 16-bit Thumb processor and using VFP, any function containing floating-point operations is compiled for ARM.
- If the architecture you are compiling code for only supports Thumb, there is no need to specify `--thumb` on the command line. For example, if compiling code for ARMv7-M with `--cpu=7-M`, you do not have to specify `--thumb` on the command line, because ARMv7-M only supports Thumb. Similarly, ARMv6-M and other Thumb-only architectures.

3.38.4 Restrictions

You cannot specify both a processor and an architecture on the same command-line.

3.38.5 See also

- [--apcs=qualifer...qualifier](#) on page 3-11
- [--cpu=list](#) on page 3-48
- [--fpu=name](#) on page 3-100
- [--thumb](#) on page 3-197

- [__smc](#) on page 5-21
- [SMC](#) on page 3-132 in the *Assembler Reference*.

3.39 --create_pch=*filename*

This option instructs the compiler to create a *PreCompiled Header* (PCH) file with the specified filename.

This option takes precedence over all other PCH options.

3.39.1 Syntax

--create_pch=*filename*

Where:

filename is the name of the PCH file to be created.

3.39.2 See also

- [--pch](#) on page 3-165
- [--pch_dir=dir](#) on page 3-166
- [--pch_messages, --no_pch_messages](#) on page 3-167
- [--pch_verbose, --no_pch_verbose](#) on page 3-168
- [--use_pch=filename](#) on page 3-211
- [#pragma hdrstop](#) on page 5-97
- [#pragma no_pch](#) on page 5-102
- [PreCompiled Header \(PCH\) files on page 5-33](#) in *Using the Compiler*.

3.40 `-Dname[(parm-list)][=def]`

This option defines the macro *name*.

3.40.1 Syntax

`-Dname[(parm-list)][=def]`

Where:

name Is the name of the macro to be defined.

parm-list Is an optional list of comma-separated macro parameters. By appending a macro parameter list to the macro name, you can define function-style macros.

The parameter list must be enclosed in parentheses. When specifying multiple parameters, do not include spaces between commas and parameter names in the list.

———— **Note** —————

Parentheses might require escaping on UNIX systems.

=def Is an optional macro definition.

If *=def* is omitted, the compiler defines *name* as the value 1.

To include characters recognized as tokens on the command line, enclose the macro definition in double quotes.

3.40.2 Usage

Specifying `-Dname` has the same effect as placing the text `#define name` at the head of each source file.

3.40.3 Restrictions

The compiler defines and undefines macros in the following order:

1. compiler predefined macros
2. macros defined explicitly, using `-Dname`
3. macros explicitly undefined, using `-Uname`.

3.40.4 Example

Specifying the option:

```
-DMAX(X,Y)="((X > Y) ? X : Y)"
```

on the command line is equivalent to defining the macro:

```
#define MAX(X, Y) ((X > Y) ? X : Y)
```

at the head of each source file.

3.40.5 See also

- [-C on page 3-32](#)
- [-E on page 3-82](#)
- [-Uname on page 3-206](#)
- [Predefined macros on page 5-183](#).

3.41 --data_reorder, --no_data_reorder

This option enables or disables automatic reordering of top-level data items, for example global variables.

The compiler can save memory by eliminating wasted space between data items. However, --data_reorder can break legacy code, if the code makes invalid assumptions about ordering of data by the compiler.

The ISO C Standard does not guarantee data order, so you must try to avoid writing code that depends on any assumed ordering. If you require data ordering, place the data items into a structure.

3.41.1 Default

The default is optimization-level dependent:

-00: --no_data_reorder

-01, -02, -03: --data_reorder

3.41.2 See also

- [-Onum on page 3-156](#)
- [Default compiler options that are affected by optimization level on page 5-45.](#)

3.42 --debug, --no_debug

This option enables or disables the generation of debug tables for the current compilation.

The compiler produces the same code regardless of whether --debug is used. The only difference is the existence of debug tables.

3.42.1 Default

The default is --no_debug.

Using --debug does not affect optimization settings. By default, using the --debug option alone is equivalent to:

```
--debug --dwarf3 --debug_macros
```

3.42.2 See also

- [--debug_macros, --no_debug_macros](#) on page 3-57
- [--dwarf2](#) on page 3-80
- [--dwarf3](#) on page 3-81
- [-Onum](#) on page 3-156.

3.43 --debug_macros, --no_debug_macros

This option enables or disables the generation of debug table entries for preprocessor macro definitions.

3.43.1 Usage

Using `--no_debug_macros` might reduce the size of the debug image.

This option must be used with the `--debug` option.

3.43.2 Default

The default is `--debug_macros`.

3.43.3 See also

- [--debug, --no_debug](#) on page 3-56
- [--gnu_defaults](#) on page 3-108.

3.44 `--default_definition_visibility=visibility`

This option controls the default ELF symbol visibility of **extern** variable and function definitions.

3.44.1 Syntax

```
--default_definition_visibility=visibility
```

Where:

visibility is default, hidden, internal, or protected.

3.44.2 Usage

Use `--default_definition_visibility=visibility` to force the compiler to use the specified ELF symbol visibility for all **extern** variables and functions defined in the source file, if they do not use `__declspec(dll*)` or `__attribute__((visibility("visibility_type")))`. Unlike `--hide_all`, `--no_hide_all`, this does not affect **extern** references.

3.44.3 Default

By default, `--default_definition_visibility=hidden`.

3.44.4 See also

- [--hide_all, --no_hide_all](#) on page 3-113
- [__attribute__\(\(visibility\("visibility_type"\)\)\) function attribute](#) on page 5-61
- [__attribute__\(\(visibility\("visibility_type"\)\)\) variable attribute](#) on page 5-81
- [Symbol visibility for BPABI models on page 10-7](#) in *Using the Linker*.

3.45 --default_extension=ext

This option enables you to change the filename extension for object files from the default extension (.o) to an extension of your choice.

3.45.1 Syntax

```
--default_extension=ext
```

Where:

`ext` is the filename extension of your choice.

3.45.2 Default

By default, the filename extension for object files is .o.

3.45.3 Example

The following example creates an object file called `test.obj`, instead of `test.o`:

```
armcc --default_extension=obj -c test.c
```

———— **Note** —————

The `-o filename` option overrides this. For example, the following command results in an object file named `test.o`:

```
armcc --default_extension=obj -o test.o -c test.c
```

3.46 --dep_name, --no_dep_name

This option enables or disables dependent name processing in C++.

The C++ standard states that lookup of names in templates occurs:

- at the time the template is parsed, if the name is nondependent
- at the time the template is parsed, or at the time the template is instantiated, if the name is dependent.

When the option `--no_dep_name` is selected, the lookup of dependent names in templates can occur only at the time the template is instantiated. That is, the lookup of dependent names at the time the template is parsed is disabled.

———— **Note** —————

The option `--no_dep_name` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

3.46.1 Mode

This option is effective only if the source language is C++.

3.46.2 Default

The default is `--dep_name`.

3.46.3 Restrictions

The option `--dep_name` cannot be combined with the option `--no_parse_templates`, because parsing is done by default when dependent name processing is enabled.

3.46.4 Errors

When the options `--dep_name` and `--no_parse_templates` are combined, the compiler generates an error.

3.46.5 See also

- [--parse_templates, --no_parse_templates on page 3-164](#)
- [Template instantiation on page 6-19](#).

3.47 `--depend=filename`

This option instructs the compiler to write makefile dependency lines to a file during compilation.

3.47.1 Syntax

`--depend=filename`

Where:

filename is the name of the dependency file to be output.

3.47.2 Usage

If you specify multiple source files on the command line then the dependency file accumulates the dependency lines from each source file. The output file is suitable for use by a make utility. To change the output format to be compatible with UNIX make utilities, use the `--depend_format` option.

3.47.3 See also

- [--depend_format=string](#) on page 3-63
- [--depend_dir=directory_name](#) on page 3-62
- [--depend_system_headers, --no_depend_system_headers](#) on page 3-66
- [--depend_target=target](#) on page 3-67
- [--ignore_missing_headers](#) on page 3-115
- [--list](#) on page 3-133
- [-M](#) on page 3-144
- [--md](#) on page 3-145
- [--depend_single_line, --no_depend_single_line](#) on page 3-65
- [--phony_targets](#) on page 3-170.

3.48 `--depend_dir=directory_name`

This option enables you to specify a directory for dependency output files.

3.48.1 Example

```
armcc -c --output_dir=obj f1.c f2.c --depend_dir=depend --depend=deps
```

Result:

```
depend/f1.d  
depend/f2.d  
obj/f1.o  
obj/f2.o
```

3.48.2 See also

- [--depend=filename](#) on page 3-61
- [--asm_dir=directory_name](#) on page 3-25
- [--list_dir=directory_name](#) on page 3-135
- [--output_dir=directory_name](#) on page 3-162.

3.49 --depend_format=string

This option changes the format of output dependency files, for compatibility with some UNIX make programs.

3.49.1 Syntax

--depend_format=string

Where *string* is one of:

unix	generate dependency file entries using UNIX-style path separators.
unix_escaped	is the same as unix, but escapes spaces with \.
unix_quoted	is the same as unix, but surrounds path names with double quotes.

3.49.2 Usage

unix	On Windows systems, --depend_format=unix forces the use of UNIX-style path names. That is, the UNIX-style path separator symbol / is used in place of \. On UNIX systems, --depend_format=unix has no effect.
unix_escaped	On Windows systems, --depend_format=unix_escaped forces unix-style path names, and escapes spaces with \. On UNIX systems, --depend_format=unix_escaped with escapes spaces with \.
unix_quoted	On Windows systems, --depend_format=unix_quoted forces unix-style path names and surrounds them with ". On UNIX systems, --depend_format=unix_quoted surrounds path names with "".

3.49.3 Default

If you do not specify a --depend_format option, then the format of output dependency files depends on your choice of operating system:

Windows	On Windows systems, the default is to use either Windows-style paths or UNIX-style paths, whichever is given.
UNIX	On UNIX systems, the default is --depend_format=unix.

3.49.4 Example

On a Windows system, compiling a file main.c containing the line:

```
#include "..\include\header files\common.h"
```

using the options --depend=depend.txt --depend_format=unix_escaped produces a dependency file depend.txt containing the entries:

```
main.axf: main.c
main.axf: ../include/header\ files/common.h
```

3.49.5 See also

- [--depend=filename](#) on page 3-61

- *--depend_system_headers*, *--no_depend_system_headers* on page 3-66
- *--depend_target=target* on page 3-67
- *--ignore_missing_headers* on page 3-115
- *-M* on page 3-144
- *--md* on page 3-145
- *--phony_targets* on page 3-170

3.50 `--depend_single_line`, `--no_depend_single_line`

This option determines the format of the makefile dependency lines output by the compiler. `--depend_single_line` instructs the compiler to format the makefile with one dependency line for each compilation unit. The compiler wraps long lines to improve readability. `--no_depend_single_line` instructs the compiler to format the makefile with one line for each include file or source file.

3.50.1 Default

The default is `--no_depend_single_line`.

3.50.2 Example

```
/* hello.c */
#include <stdio.h>
int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

Compiling this code with `armcc hello.c -M --depend_single_line` produces:

```
__image.axf: hello.c...\include...\stdio.h
```

Compiling this code with `armcc hello.c -M --no_depend_single_line` produces:

```
__image.axf: hello.c
__image.axf: ...\include...\stdio.h
```

3.50.3 See also

- [--depend=filename](#) on page 3-61
- [--depend_format=string](#) on page 3-63
- [--depend_target=target](#) on page 3-67
- [--ignore_missing_headers](#) on page 3-115
- [-M](#) on page 3-144
- [--md](#) on page 3-145
- [--phony_targets](#) on page 3-170.

3.51 --depend_system_headers, --no_depend_system_headers

This option enables or disables the output of system include dependency lines when generating makefile dependency information using either the `-M` option or the `--md` option.

3.51.1 Default

The default is `--depend_system_headers`.

3.51.2 Example

```
/* hello.c */
#include <stdio.h>
int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

Compiling this code with the option `-M` produces:

```
__image.axf: hello.c
__image.axf: ... \include\... \stdio.h
```

Compiling this code with the options `-M --no_depend_system_headers` produces:

```
__image.axf: hello.c
```

3.51.3 See also

- [--depend=filename](#) on page 3-61
- [--depend_format=string](#) on page 3-63
- [--depend_target=target](#) on page 3-67
- [--ignore_missing_headers](#) on page 3-115
- [-M](#) on page 3-144
- [--md](#) on page 3-145
- [--phony_targets](#) on page 3-170

3.52 `--depend_target=target`

This option sets the target for makefile dependency generation.

3.52.1 Usage

Use this option to override the default target.

3.52.2 Restriction

This option is analogous to `-MT` in GCC. However, behavior differs when specifying multiple targets. For example, `gcc -M -MT target1 -MT target2 file.c` might give a result of `target1 target2: file.c header.h`, whereas `--depend_target=target1 --depend_target=target2` treats `target2` as the target.

3.52.3 See also

- [--depend=filename](#) on page 3-61
- [--depend_format=string](#) on page 3-63
- [--depend_system_headers](#), [--no_depend_system_headers](#) on page 3-66
- [--ignore_missing_headers](#) on page 3-115
- [-M](#) on page 3-144
- [--md](#) on page 3-145
- [--phony_targets](#) on page 3-170

3.53 --device=list

This option lists the supported device names that can be used with the `--device=name` option.

———— **Note** —————

This option is deprecated.

3.53.1 See also

- [--device=name](#) on page 3-69.

3.54 --device=*name*

This option enables you to compile code for a specific microcontroller or *System-on-Chip* (SoC) device.

———— **Note** —————

This option is deprecated.

3.54.1 Syntax

--device=*name*

Where:

name is the name of a target microcontroller or SoC device.

3.54.2 Usage

When you specify a particular device name, the device inherits the default endianness and floating-point architecture from the corresponding CPU. You can use the --bi, --li, and --fpu options to alter the default settings for endianness and target floating-point architecture.

3.54.3 See also

- [--bigend](#) on page 3-27
- [--device=list](#) on page 3-68
- [--fpu=name](#) on page 3-100
- [--littleend](#) on page 3-137
- [--device=list](#) on page 2-42 in the *Linker Reference*
- [--device=name](#) on page 2-43 in the *Linker Reference*
- [Using the C preprocessor](#) on page 7-24 in *Using the Assembler*.

3.55 `--diag_error=tag[, tag, ...]`

This option sets diagnostic messages that have a specific tag to error severity.

Note

This option has the `#pragma` equivalent `#pragma diag_error`.

Note

For some diagnostic message numbers in the range 3000-3100, the same number can refer to two unrelated messages. ARM will address this in a future release. If you use this option, specifying such a number, it affects both messages.

3.55.1 Syntax

`--diag_error=tag[, tag, ...]`

Where *tag* can be:

- a diagnostic message number to set to error severity
- `warning`, to treat all warnings as errors.

3.55.2 Usage

The severity of the following types of diagnostic messages can be changed:

- Messages with the number format `#nnnn-D`.
- Warning messages with the number format `CnnnnW`.

3.55.3 See also

- [--diag_remark=tag\[,tag,...\]](#) on page 3-71
- [--diag_suppress=tag\[,tag,...\]](#) on page 3-73
- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [#pragma diag_error tag\[,tag,...\]](#) on page 5-91
- [Options that change the severity of compiler diagnostic messages on page 7-4](#) in *Using the Compiler*.

3.56 `--diag_remark=tag[, tag, ...]`

This option sets the diagnostic messages that have the specified tags to Remark severity.

The `--diag_remark` option behaves analogously to `--diag_errors`, except that the compiler sets the diagnostic messages having the specified tags to Remark severity rather than Error severity.

Note

Remarks are not displayed by default. To see remark messages, use the compiler option `--remarks`.

Note

This option has the `#pragma` equivalent `#pragma diag_remark`.

Note

For some diagnostic message numbers in the range 3000-3100, the same number can refer to two unrelated messages. ARM will address this in a future release. If you use this option, specifying such a number, it affects both messages.

3.56.1 Syntax

`--diag_remark=tag[, tag, ...]`

Where:

`tag[, tag, ...]` is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.

3.56.2 See also

- [--diag_error=tag\[,tag,...\]](#) on page 3-70
- [--diag_suppress=tag\[,tag,...\]](#) on page 3-73
- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [--remarks](#) on page 3-181
- [#pragma diag_remark tag\[,tag,...\]](#) on page 5-92
- [Options that change the severity of compiler diagnostic messages on page 7-4](#) in *Using the Compiler*.

3.57 `--diag_style={arm|ide|gnu}`

This option specifies the style used to display diagnostic messages.

3.57.1 Syntax

`--diag_style=string`

Where *string* is one of:

arm	Display messages using the ARM compiler style.
ide	Include the line number and character count for any line that is in error. These values are displayed in parentheses.
gnu	Display messages in the format used by gcc.

3.57.2 Default

If you do not specify a `--diag_style` option, the compiler assumes `--diag_style=arm`.

3.57.3 Usage

Choosing the option `--diag_style=ide` implicitly selects the option `--brief_diagnostics`. Explicitly selecting `--no_brief_diagnostics` on the command line overrides the selection of `--brief_diagnostics` implied by `--diag_style=ide`.

Selecting either the option `--diag_style=arm` or the option `--diag_style=gnu` does not imply any selection of `--brief_diagnostics`.

3.57.4 See also

- [--diag_error=tag\[,tag,...\]](#) on page 3-70
- [--diag_remark=tag\[,tag,...\]](#) on page 3-71
- [--diag_suppress=tag\[,tag,...\]](#) on page 3-73
- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [Options that change the severity of compiler diagnostic messages on page 7-4](#) in *Using the Compiler*.

3.58 --diag_suppress=tag[, tag, ...]

This option disables diagnostic messages that have the specified tags.

The --diag_suppress option behaves analogously to --diag_errors, except that the compiler suppresses the diagnostic messages having the specified tags rather than setting them to have error severity.

———— **Note** —————

This option has the #pragma equivalent #pragma diag_suppress.

———— **Note** —————

For some diagnostic message numbers in the range 3000-3100, the same number can refer to two unrelated messages. ARM will address this in a future release. If you use this option, specifying such a number, it affects both messages.

3.58.1 Syntax

--diag_suppress=tag[, tag, ...]

Where *tag* can be:

- a diagnostic message number to be suppressed
- error, to suppress all downgradable errors
- warning, to suppress all warnings.

3.58.2 See also

- [--diag_error=tag\[,tag,...\]](#) on page 3-70
- [--diag_remark=tag\[,tag,...\]](#) on page 3-71
- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [#pragma diag_suppress tag\[,tag,...\]](#) on page 5-93
- [About compiler diagnostic messages on page 7-2](#) in *Using the Compiler*
- [Prefix letters in compiler diagnostic messages on page 7-5](#) in *Using the Compiler*.

3.59 --diag_suppress=optimizations

This option suppresses diagnostic messages for high-level optimizations.

3.59.1 Default

By default, optimization messages have Remark severity. Specifying `--diag_suppress=optimizations` suppresses optimization messages.

———— **Note** —————

Use the `--remarks` option to see optimization messages having Remark severity.

3.59.2 Usage

The compiler performs certain high-level vector and scalar optimizations when compiling at the optimization level `-O3 -Otime`, for example, loop unrolling. Use this option to suppress diagnostic messages relating to these high-level optimizations.

3.59.3 Example

```
int factorial(int n)
{
    int result=1;
    while (n > 0)
        result *= n--;
    return result;
}
```

Compiling this code with the options `-O3 -Otime --remarks --diag_suppress=optimizations` suppresses optimization messages.

3.59.4 See also

- [--diag_suppress=tag\[,tag,...\]](#) on page 3-73
- [--diag_warning=optimizations](#) on page 3-76
- [-Onum](#) on page 3-156
- [-Otime](#) on page 3-161
- [--remarks](#) on page 3-181.

3.60 `--diag_warning=tag[, tag, ...]`

This option sets diagnostic messages that have the specified tags to warning severity.

The `--diag_warning` option behaves analogously to `--diag_errors`, except that the compiler sets the diagnostic messages having the specified tags to warning severity rather than error severity.

Note

This option has the `#pragma` equivalent `#pragma diag_warning`.

Note

For some diagnostic message numbers in the range 3000-3100, the same number can refer to two unrelated messages. ARM will address this in a future release. If you use this option, specifying such a number, it affects both messages.

3.60.1 Syntax

`--diag_warning=tag[, tag, ...]`

Where *tag* can be:

- a diagnostic message number to set to warning severity
- `error`, to downgrade the severity of all downgradable errors to warnings.

3.60.2 Example

`--diag_warning=A1234,error` causes message A1234 and all downgradable errors to be treated as warnings, providing changing the severity of A1234 is permitted.

3.60.3 See also

- [--diag_error=tag\[,tag,...\]](#) on page 3-70
- [--diag_remark=tag\[,tag,...\]](#) on page 3-71
- [--diag_suppress=tag\[,tag,...\]](#) on page 3-73
- [#pragma diag_warning tag\[, tag, ...\]](#) on page 5-94
- [Options that change the severity of compiler diagnostic messages on page 7-4](#) in *Using the Compiler*.

3.61 `--diag_warning=optimizations`

This option sets high-level optimization diagnostic messages to have Warning severity.

3.61.1 Default

By default, optimization messages have Remark severity.

3.61.2 Usage

The compiler performs certain high-level vector and scalar optimizations when compiling at the optimization level `-O3 -Otime`, for example, loop unrolling. Use this option to display diagnostic messages relating to these high-level optimizations.

3.61.3 Example

```
int factorial(int n)
{
    int result=1;
    while (n > 0)
        result *= n--;
    return result;
}
```

Compiling this code with the options `--vectorize --cpu=Cortex-A8 -O3 -Otime --diag_warning=optimizations` generates optimization warning messages.

3.61.4 See also

- [--diag_suppress=optimizations](#) on page 3-74
- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [-Onum](#) on page 3-156
- [-Otime](#) on page 3-161.

3.62 `--dllexport_all`, `--no_dllexport_all`

This option enables you to control symbol visibility when building DLLs.

3.62.1 Default

The default is `--no_dllexport_all`.

3.62.2 Usage

Use the option `--dllexport_all` to mark all extern definitions as `__declspec(dllexport)`.

3.62.3 See also

- [--apcs=qualifer...qualifier](#) on page 3-11
- [__declspec\(dllexport\)](#) on page 5-31.

3.63 `--dllimport_runtime, --no_dllimport_runtime`

This option enables you to control symbol visibility when using the runtime library as a shared library.

3.63.1 Default

The default is `--no_dllimport_runtime`.

3.63.2 Usage

Use the option `--dllimport_runtime` to mark all implicit references as `__declspec(dllimport)`. Implicit references are references that are not in user source code but are nonetheless used by the compiler. Implicit references include:

- Library-resident compiler helper functions. For example, helper functions for software floating-point support.
- *RunTime Type Information* (RTTI) found in the C++ runtime libraries.
- Any optimized implementation of a user-specified function, for example, `printf()`, providing that the non-optimized user-specified version of the function that the optimized implementation is based on, is marked as `__declspec(dllimport)`. Header files describing which library functions are exported from DLLs are usually provided with the platform DLL version of the C library.

3.63.3 See also

- [--guiding_decls, --no_guiding_decls](#) on page 3-111
- [--rtti, --no_rtti](#) on page 3-185
- [__declspec\(dllimport\)](#) on page 5-33.

3.64 --dollar, --no_dollar

This option instructs the compiler to accept or reject dollar signs, \$, in identifiers.

3.64.1 Default

If the options `--strict` or `--strict_warnings` are specified, the default is `--no_dollar`. Otherwise, the default is `--dollar`.

3.64.2 See also

- [Dollar signs in identifiers on page 4-21](#)
- [--strict, --no_strict on page 3-194](#).

3.65 --dwarf2

This option instructs the compiler to use DWARF 2 debug table format.

3.65.1 Default

The compiler assumes `--dwarf3` unless `--dwarf2` is explicitly specified.

3.65.2 See also

- [--dwarf3](#) on page 3-81.

3.66 --dwarf3

This option instructs the compiler to use DWARF 3 debug table format.

3.66.1 Default

The compiler assumes --dwarf3 unless --dwarf2 is explicitly specified.

3.66.2 See also

- [--dwarf2](#) on page 3-80.

3.67 -E

This option instructs the compiler to execute only the preprocessor step.

By default, output from the preprocessor is sent to the standard output stream and can be redirected to a file using standard UNIX and MS-DOS notation.

You can also use the `-o` option to specify a file for the preprocessed output. By default, comments are stripped from the output. The preprocessor accepts source files with any extension, for example, `.o`, `.s`, and `.txt`.

To generate interleaved macro definitions and preprocessor output, use `-E --list_macros`.

Note

C++ implicit inclusion does not take place when using the `armcc -E` preprocessor. Normally, compilation expands all explicit `#include` header files. In addition, some C++ files such as `.cc` files are added implicitly. However, using `-E` prevents implicit inclusion of these files. Therefore, if template entities are defined in a `.cc` file, `armcc -E` will fail to include such definitions.

3.67.1 Example

```
armcc -E source.c > raw.c
```

3.67.2 See also

- [-C on page 3-32](#)
- [--list_macros on page 3-136](#)
- [--md on page 3-145](#)
- [-o filename on page 3-154](#)
- [--old_style_preprocessing on page 3-159](#)
- [-P on page 3-163](#)
- [Why does "armcc -E" preprocessing result in linker undefined symbol error?,
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka14374.html](#).

3.68 --echo

Instructs the compiler to display the complete expanded command-line it uses, and any separate commands that it uses to invoke other external applications, such as `armasm` or `armlink`.

This command is useful when specifying options that cause multiple command invocations, such as GCC fallback.

3.68.1 Usage

If you use `--echo` when performing GCC fallback, you must specify it using `-Warmcc,-echo`.

3.68.2 Examples

To compile and link:

```
armcc --echo foo.c -o foo.axf
[armcc --echo -ofoo.axf foo.c]
[armlink -o foo.axf foo.o --fpu=SoftVFP --li]
```

To compile only:

```
armcc -c --echo foo.c -o foo.axf
[armcc --echo -c -ofoo.axf foo.c]
```

3.68.3 See also

- [-Warmcc,option\[,option,...\]](#) on page 3-221
- [-Warmcc,--gcc_fallback](#) on page 3-222.

3.69 `--emit_frame_directives`, `--no_emit_frame_directives`

This option instructs the compiler to place DWARF FRAME directives into disassembly output.

3.69.1 Default

The default is `--no_emit_frame_directives`.

3.69.2 Examples

```
armcc --asm --emit_frame_directives foo.c
```

```
armcc -S emit_frame_directives foo.c
```

3.69.3 See also

- [--asm on page 3-24](#)
- [-S on page 3-187](#)
- [Frame directives on page 5-37](#) in *Using the Assembler*.

3.70 --enum_is_int

This option forces the size of all enumeration types to be at least four bytes.

———— **Note** —————

The --enum_is_int option is not recommended for general use.

3.70.1 Default

This option is switched off by default. The smallest data type that can hold the values of all enumerators is used. However, if you specify an ARM Linux configuration file on the command line, --enum_is_int is switched on by default.

3.70.2 See also

- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--interface_enums_are_32_bit](#) on page 3-123
- [Structures, unions, enumerations, and bitfields](#) on page 6-9.

3.71 `--errors=filename`

This option redirects the output of diagnostic messages from `stderr` to the specified errors file.

3.71.1 Syntax

```
--errors=filename
```

Where:

filename is the name of the file to which errors are to be redirected.

Diagnostics that relate to problems with the command options are not redirected, for example, if you type an option name incorrectly. However, if you specify an invalid argument to an option, for example `--cpu=999`, the related diagnostic is redirected to the specified *filename*.

3.71.2 Usage

This option is useful on systems where output redirection of files is not well supported.

3.71.3 See also

- [--brief_diagnostics, --no_brief_diagnostics](#) on page 3-29
- [--diag_error=tag\[,tag,...\]](#) on page 3-70
- [--diag_remark=tag\[,tag,...\]](#) on page 3-71
- [--diag_style={arm|ide|gnu}](#) on page 3-72
- [--diag_suppress=tag\[,tag,...\]](#) on page 3-73
- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [--remarks](#) on page 3-181
- [-W](#) on page 3-220
- [--wrap_diagnostics, --no_wrap_diagnostics](#) on page 3-228
- [Chapter 7 Compiler Diagnostic Messages](#) in *Using the Compiler*.

3.72 --exceptions, --no_exceptions

This option enables or disables exception handling.

In C++, the `--exceptions` option enables the use of `throw` and `try/catch`, causes function exception specifications to be respected, and causes the compiler to emit unwinding tables to support exception propagation at runtime.

In C++, when the `--no_exceptions` option is specified, `throw` and `try/catch` are not permitted in source code. However, function exception specifications are still parsed, but most of their meaning is ignored.

In C, the behavior of code compiled with `--no_exceptions` is undefined if an exception is thrown through the compiled functions. You must use `--exceptions`, if you want exceptions to propagate correctly through C functions.

3.72.1 Default

The default is `--no_exceptions`. However, if you specify an ARM Linux configuration file on the command line and you use `--translate_g++`, the default changes to `--exceptions`.

3.72.2 See also

- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--exceptions_unwind, --no_exceptions_unwind](#) on page 3-88.

3.73 `--exceptions_unwind`, `--no_exceptions_unwind`

This option enables or disables function unwinding for exception-aware code. This option is only effective if `--exceptions` is enabled.

When you use `--no_exceptions_unwind` and `--exceptions` then no exception can propagate through the compiled functions. `std::terminate` is called instead.

3.73.1 Default

The default is `--exceptions_unwind`.

3.73.2 See also

- [--exceptions, --no_exceptions on page 3-87](#)
- [Function unwinding at runtime on page 6-22.](#)

3.74 --execstack, --no_execstack

--execstack generates a .note.GNU-stack section marking the stack as executable.

--no_execstack generates a .note.GNU-stack section marking the stack as non-executable.

If this option is not used, the note section is not generated.

--arm_linux implies --no_execstack, unless --execstack is explicitly specified.

3.74.1 See also

- [--arm_linux](#) on page 3-16.

3.75 `--export_all_vtbl`, `--no_export_all_vtbl`

This option controls how dynamic symbols are exported in C++.

3.75.1 Mode

This option is effective only if the source language is C++.

3.75.2 Default

The default is `--no_export_all_vtbl`.

3.75.3 Usage

Use the option `--export_all_vtbl` to export all virtual function tables and RTTI for classes with a key function. A *key function* is the first virtual function of a class, in declaration order, that is not inline, and is not pure virtual.

———— **Note** —————

You can disable export for specific classes by using `__declspec(notshared)`.

3.75.4 See also

- [__declspec\(notshared\)](#) on page 5-37
- [--import_all_vtbl](#) on page 3-120.

3.76 `--export_defs_implicitly`, `--no_export_defs_implicitly`

This option controls how dynamic symbols are exported.

3.76.1 Default

The default is `--no_export_defs_implicitly`.

3.76.2 Usage

Use the option `--export_defs_implicitly` to export definitions where the prototype is marked `__declspec(dllimport)`.

3.76.3 See also

- [__declspec\(dllimport\)](#) on page 5-33.

3.77 `--extended_initializers`, `--no_extended_initializers`

These options enable and disable the use of extended constant initializers even when compiling with `--strict` or `--strict_warnings`.

When certain nonportable but widely supported constant initializers such as the cast of an address to an integral type are used, `--extended_initializers` causes the compiler to produce the same general warning concerning constant initializers that it normally produces in nonstrict mode, rather than specific errors stating that the expression must have a constant value or have arithmetic type.

3.77.1 Default

The default is `--no_extended_initializers` when compiling with `--strict` or `--strict_warnings`.

The default is `--extended_initializers` when compiling in nonstrict mode.

3.77.2 See also

- [--strict, --no_strict](#) on page 3-194
- [--strict_warnings](#) on page 3-195
- [Constant expressions](#) on page 4-18.

3.78 `--feedback=filename`

This option enables the efficient elimination of unused functions, and on the ARMv4T architecture, enables reduction of compilation required for interworking.

3.78.1 Syntax

`--feedback=filename`

Where:

filename is the feedback file created by a previous execution of the ARM linker.

3.78.2 Usage

You can perform multiple compilations using the same feedback file. The compiler places each unused function identified in the feedback file into its own ELF section in the corresponding object file.

The feedback file contains information about a previous build. Because of this:

- The feedback file might be out of date. That is, a function previously identified as being unused might be used in the current source code. The linker removes the code for an unused function only if it is not used in the current source code.

———— **Note** —————

- For this reason, eliminating unused functions using linker feedback is a safe optimization, but there might be a small impact on code size.
- The usage requirements for reducing compilation required for interworking are more strict than for eliminating unused functions. If you are reducing interworking compilation, it is critical that you keep your feedback file up to date with the source code that it was generated from.

- You have to do a full compile and link at least twice to get the maximum benefit from linker feedback. However, a single compile and link using feedback from a previous build is usually sufficient.

3.78.3 See also

- [--split_sections](#) on page 3-193
- [--feedback_type=type](#) on page 2-68 in the *Linker Reference*
- [Linker feedback during compilation on page 3-23](#) in *Using the Compiler*.

3.79 --force_new_nothrow, --no_force_new_nothrow

This option controls the behavior of new expressions in C++.

The C++ standard states that only a no throw operator new declared with throw() is permitted to return NULL on failure. Any other operator new is never permitted to return NULL and the default operator new throws an exception on failure.

If you use --force_new_nothrow, the compiler treats expressions such as new T(...args...), that use the global ::operator new or ::operator new[], as if they are new (std::nothrow) T(...args...).

--force_new_nothrow also causes any class-specific operator new or any overloaded global operator new to be treated as no throw.

———— **Note** —————

The option --force_new_nothrow is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

3.79.1 Mode

This option is effective only if the source language is C++.

3.79.2 Default

The default is --no_force_new_nothrow.

3.79.3 Example

```
struct S
{
    void* operator new(std::size_t);
    void* operator new[](std::size_t);
};
void *operator new(std::size_t, int);
```

With the --force_new_nothrow option in effect, this is treated as:

```
struct S
{
    void* operator new(std::size_t) throw();
    void* operator new[](std::size_t) throw();
};
void *operator new(std::size_t, int) throw();
```

3.79.4 See also

- [Using the ::operator new function on page 6-15.](#)

3.80 --forceinline

This option forces all inline functions to be treated as if they are qualified with `__forceinline`.

Inline functions are functions that are qualified with `inline` or `__inline`. In C++, inline functions are functions that are defined inside a struct, class, or union definition.

If you use `--forceinline`, the compiler always attempts to inline those functions, if possible. However, it does not inline a function if doing so causes problems. For example, a recursive function is inlined into itself only once.

`__forceinline` behaves like `__inline` except that the compiler tries harder to do the inlining.

3.80.1 See also

- [--autoinline, --no_autoinline](#) on page 3-26
- [--inline, --no_inline](#) on page 3-122
- [__forceinline](#) on page 5-10
- [__inline](#) on page 5-13
- [Inline functions](#) on page 6-30 in *Using the Compiler*.

3.81 `--fp16_format=format`

This option enables the use of half-precision floating-point numbers as an optional extension to the VFPv3 architecture. If a format is not specified, use of the `__fp16` data type is faulted by the compiler.

3.81.1 Syntax

`--fp16_format=format`

Where *format* is one of:

- | | |
|-------------|---|
| alternative | An alternative to <code>ieee</code> that provides additional range, but has no NaN or infinity values. |
| ieee | Half-precision binary floating-point format defined by IEEE 754r, a revision to the IEEE 754 standard. |
| none | This is the default setting. It is equivalent to not specifying a format and means that the compiler faults use of the <code>__fp16</code> data type. |

3.81.2 Restrictions

The following restrictions apply when you use the `__fp16` data type:

- When used in a C or C++ expression, an `__fp16` type is promoted to single precision. Subsequent promotion to double precision can occur if required by one of the operands.
- A single precision value can be converted to `__fp16`. A double precision value is converted to single precision and then to `__fp16`, that could involve double rounding. This reflects the lack of direct double-to-16-bit conversion in the ARM architecture.
- When using `fpmode=fast`, no floating-point exceptions are raised when converting to and from half-precision floating-point format.
- Function formal arguments cannot be of type `__fp16`. However, pointers to variables of type `__fp16` can be used as function formal argument types.
- `__fp16` values can be passed as actual function arguments. In this case, they are converted to single-precision values.
- `__fp16` cannot be specified as the return type of a function. However, a pointer to an `__fp16` type can be used as a return type.
- An `__fp16` value is converted to a single-precision or double-precision value when used as a return value for a function that returns a `float` or `double`.

3.81.3 See also

- [--fpmode=model](#) on page 3-97
- [Intrinsics](#) on page G-10
- [Compiler and library support for half-precision floating-point numbers](#) on page 6-63 of *Using the Compiler*.

3.82 --fpmode=*mode*

This option specifies the floating-point conformance, and sets library attributes and floating-point optimizations.

3.82.1 Syntax

--fpmode=*mode*

Where *mode* is one of:

ieee_full All facilities, operations, and representations guaranteed by the IEEE standard are available in single and double-precision. Modes of operation can be selected dynamically at runtime.

This defines the symbols:

```
__FP_IEEE
__FP_FENV_EXCEPTIONS
__FP_FENV_ROUNDING
__FP_INEXACT_EXCEPTION
```

ieee_fixed

IEEE standard with round-to-nearest and no inexact exceptions.

This defines the symbols:

```
__FP_IEEE
__FP_FENV_EXCEPTIONS
```

ieee_no_fenv

IEEE standard with round-to-nearest and no exceptions. This mode is stateless and is compatible with the Java floating-point arithmetic model.

This defines the symbol `__FP_IEEE`.

none

The compiler permits `--fpmode=none` as an alternative to `--fpu=none`, indicating that source code is not permitted to use floating-point types of any kind.

std

IEEE finite values with denormals flushed to zero, round-to-nearest, and no exceptions. This is compatible with standard C and C++ and is the default option.

Normal finite values are as predicted by the IEEE standard. However:

- NaNs and infinities might not be produced in all circumstances defined by the IEEE model. When they are produced, they might not have the same sign.
- The sign of zero might not be that predicted by the IEEE model.

Using a NaN with `--fpmode=std` can produce undefined behavior.

fast

Perform more aggressive floating-point optimizations that might cause a small loss of accuracy to provide a significant performance increase. This option defines the symbol `__FP_FAST`.

This option results in behavior that is not fully compliant with the ISO C or C++ standard. However, numerically robust floating-point programs are expected to behave correctly.

A number of transformations might be performed, including:

- Double-precision math functions might be converted to single precision equivalents if all floating-point arguments can be exactly represented as single precision values, and the result is immediately converted to a single-precision value.

This transformation is only performed when the selected library contains the single-precision equivalent functions, for example, when the selected library is `armcc` or `aeabi_glibc`.

For example:

```
float f(float a)
{
    return sqrt(a);
}
```

is transformed to

```
float f(float a)
{
    return sqrtf(a);
}.
```

- Double-precision floating-point expressions that are narrowed to single-precision are evaluated in single-precision when it is beneficial to do so. For example, `float y = (float)(x + 1.0)` is evaluated as `float y = (float)x + 1.0f`.
- Division by a floating-point constant is replaced by multiplication with the inverse. For example, `x / 3.0` is evaluated as `x * (1.0 / 3.0)`.
- It is not guaranteed that the value of `errno` is compliant with the ISO C or C++ standard after math functions have been called. This enables the compiler to inline the VFP square root instructions in place of calls to `sqrt()` or `sqrtf()`.

Using a NaN with `--fpmode=fast` can produce undefined behavior.

Note

Initialization code might be required to enable the VFP. See [Limitations on hardware handling of floating-point arithmetic on page 6-61](#) in *Using the Compiler* for more information.

3.82.2 Default

By default, `--fpmode=std` applies.

3.82.3 See also

- [--fpu=name](#) on page 3-100
- *Using VFP with RVDS, Application Note 133*, <http://infocenter/help/index.jsp?topic=/com.arm.doc.dai0133->

3.83 --fpu=list

This option lists the supported FPU architecture names that you can use with the `--fpu=name` option.

Deprecated options are not listed.

3.83.1 See also

- [--fpu=name](#) on page 3-100.

3.84 --fpu=*name*

This option enables you to specify the target FPU architecture.

If you specify this option, it overrides any implicit FPU option that appears on the command line, for example, where you use the `--cpu` option.

To obtain a full list of FPU architectures use the `--fpu=list` option.

3.84.1 Syntax

`--fpu=name`

Where *name* is one of:

- none Selects no floating-point option. No floating-point code is to be used. This produces an error if your code contains **float** types.
- vfp This is a synonym for `vfpv2`.
- vfpv2 Selects a hardware vector floating-point unit conforming to architecture VFPv2.

———— **Note** ————

If you enter `armcc --thumb --fpu=vfpv2` on the command line, the compiler compiles as much of the code using the Thumb instruction set as possible, but hard floating-point sensitive functions are compiled to ARM code. In this case, the value of the predefine `__thumb` is not correct.

- vfpv3 Selects a hardware vector floating-point unit conforming to architecture VFPv3. VFPv3 is backwards compatible with VFPv2 except that VFPv3 cannot trap floating-point exceptions.
- vfpv3_fp16 Selects a hardware vector floating-point unit conforming to architecture VFPv3 that also provides the half-precision extensions.
- vfpv3_d16 Selects a hardware vector floating-point unit conforming to VFPv3-D16 architecture.
- vfpv3_d16_fp16 Selects a hardware vector floating-point unit conforming to VFPv3-D16 architecture, that also provides the half-precision extensions.
- vfpv4 Selects a hardware floating-point unit conforming to the VFPv4 architecture.
- vfpv4_d16 Selects a hardware floating-point unit conforming to the VFPv4-D16 architecture.
- fpv4-sp Selects a hardware floating-point unit conforming to the single precision variant of the FPv4 architecture.
- softvfp Selects software floating-point support where floating-point operations are performed by a floating-point library, `fp1ib`. This is the default if you do not specify a `--fpu` option, or if you select a CPU that does not have an FPU.
- softvfp+vfpv2 Selects a hardware vector floating-point unit conforming to VFPv2, with software floating-point linkage. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFP unit.

If you select this option:

- Compiling with `--thumb` behaves in a similar way to `--fpu=softvfp` except that it links with floating-point libraries that use VFP instructions.
- Compiling with `--arm` behaves in a similar way to `--fpu=vfpv2` except that all functions are given software floating-point linkage. This means that functions pass and return floating-point arguments and results in the same way as `--fpu=softvfp`, but use VFP instructions internally.

Note

If you specify `softvfp+vfpv2` with the `--arm` or `--thumb` option for C code, it ensures that your interworking floating-point code is compiled to use software floating-point linkage.

`softvfp+vfpv3`

Selects a hardware vector floating-point unit conforming to VFPv3, with software floating-point linkage. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFPv3 unit.

`softvfp+vfpv3_fp16`

Selects a hardware vector floating-point unit conforming to VFPv3-fp16, with software floating-point linkage.

`softvfp+vfpv3_d16`

Selects a hardware vector floating-point unit conforming to VFPv3-D16, with software floating-point linkage.

`softvfp+vfpv3_d16_fp16`

Selects a hardware vector floating-point unit conforming to VFPv3-D16-fp16, with software floating-point linkage.

`softvfp+vfpv4`

Selects a hardware floating-point unit conforming to FPv4, with software floating-point linkage.

`softvfp+vfpv4_d16`

Selects a hardware floating-point unit conforming to VFPv4-D16, with software floating-point linkage.

`softvfp+fpv4-sp`

Selects a hardware floating-point unit conforming to FPv4-SP, with software floating-point linkage.

3.84.2 Usage

Any FPU explicitly selected using the `--fpu` option always overrides any FPU implicitly selected using the `--cpu` option. For example, the option `--cpu=ARM1136JF-S --fpu=softvfp` generates code that uses the software floating-point library `fp1ib`, even though the choice of CPU implies the use of architecture VFPv2.

To control floating-point linkage without affecting the choice of FPU, you can use `--apcs=/softfp` or `--apcs=/hardfp`.

3.84.3 Restrictions

The compiler only permits hardware VFP architectures (for example, `--fpu=vfpv3`, `--fpu=softvfp+vfpv2`), to be specified when MRRC and MCRR instructions are supported in the processor instruction set. MRRC and MCRR instructions are not supported in 4, 4T, 5T and 6-M. Therefore, the compiler does not allow the use of these CPU architectures with hardware VFP architectures.

Other than this, the compiler does not check that `--cpu` and `--fpu` combinations are valid. Beyond the scope of the compiler, additional architectural constraints apply. For example, VFPv3 is not supported with architectures prior to ARMv7. Therefore, the combination of `--fpu` and `--cpu` options permitted by the compiler does not necessarily translate to the actual device in use.

The compiler only generates scalar floating-point operations. If you want to use VFP vector operations, you must do this using assembly code.

NEON support is disabled for `softvfp`.

3.84.4 Default

The default target FPU architecture is derived from use of the `--cpu` option.

If the CPU specified with `--cpu` has a VFP coprocessor, the default target FPU architecture is the VFP architecture for that CPU. For example, the option `--cpu ARM1136JF-S` implies the option `--fpu vfpv2`. If a VFP coprocessor is present, VFP instructions are generated.

If you are building ARM Linux applications with `--arm_linux` or `--arm_linux_paths`, the default is always software floating-point linkage. Even if you specify a CPU that implies an FPU (for example, `--cpu=ARM1136JF-S`), the compiler still defaults to `--fpu=softvfp+vfp`, not `--fpu=vfp`.

If there is no VFP coprocessor, the compiler generates code that makes calls to the software floating-point library `fp1ib` to carry out floating-point operations.

3.84.5 See also

- [--apcs=qualifer...qualifier](#) on page 3-11
- [--arm](#) on page 3-15
- [--cpu=name](#) on page 3-49
- [--fpmode=model](#) on page 3-97
- [--thumb](#) on page 3-197
- [__softfp](#) on page 5-22
- [Vector Floating-Point \(VFP\) architectures](#) on page 6-60 in *Using the Compiler*
- [Compiler support for floating-point computations and linkage](#) on page 6-65 in *Using the Compiler*
- [ARM and Thumb floating-point build options \(ARMv6 and earlier\)](#) on page 2-9 in *Developing Software for ARM Processors*
- [ARM and Thumb floating-point build options \(ARMv7 and later\)](#) on page 2-10 in *Developing Software for ARM Processors*
- [MRC, MRC2, MRRC and MRRC2](#) on page 3-102 in the *Assembler Reference*

- *MCR, MCR2, MCRR, and MCRR2 on page 3-94* in the *Assembler Reference*.

3.85 `--friend_injection`, `--no_friend_injection`

This option controls the visibility of friend declarations in C++.

In C++, it controls whether or not the name of a class or function that is declared only in friend declarations is visible when using the normal lookup mechanisms.

When friend names are declared, they are visible to these lookups. When friend names are not declared as required by the standard, function names are visible only when using argument-dependent lookup, and class names are never visible.

———— **Note** —————

The option `--friend_injection` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

3.85.1 Mode

This option is effective only if the source language is C++.

3.85.2 Default

The default is `--no_friend_injection`.

3.85.3 See also

- [friend](#) on page 4-29.

3.86 -g

This option enables the generation of debug tables for the current compilation.

The compiler produces the same code regardless of whether `-g` is used. The only difference is the existence of debug tables.

Using `-g` does not affect optimization settings.

3.86.1 Default

By default, using the `-g` option alone is equivalent to:

```
-g --dwarf3 --debug_macros
```

3.86.2 See also

- [--debug, --no_debug](#) on page 3-56
- [--debug_macros, --no_debug_macros](#) on page 3-57
- [--dwarf2](#) on page 3-80
- [--dwarf3](#) on page 3-81
- [-Onum](#) on page 3-156.

3.87 `--global_reg=reg_name[, reg_name, ...]`

This option treats the specified register names as fixed registers, and prevents the compiler from using them in the code that is generated.

Note

Try to avoid using this option, because it restricts the compiler in terms of register allocation and can potentially give a negative effect on code generation and performance.

3.87.1 Syntax

`--global_reg=reg_name[, reg_name, ...]`

Where *reg_name* is the AAPCS name of the register, denoted by an integer value in the range 1 to 8.

Register names 1 to 8 map sequentially onto registers r4 to r11.

If *reg_name* is unspecified, the compiler faults use of `--global_reg`.

3.87.2 Restrictions

This option has the same restrictions as the `__global_reg` storage class specifier.

3.87.3 Example

`--global_reg=1,4,5 // reserve registers r4, r7 and r8 respectively`

3.87.4 See also

- [__global_reg](#) on page 5-11
- *ARM Software Development Toolkit Reference Guide*.

3.88 --gnu

This option enables the GNU compiler extensions supported by the ARM compiler. The version of GCC the extensions are compatible with can be determined by inspecting the predefined macros `__GNUC__` and `__GNUC_MINOR__`.

In addition, in GNU mode, the ARM compiler emulates GCC in its conformance to the C/C++ standards, whether more or less strict.

3.88.1 Usage

This option can also be combined with other source language command-line options. For example, `armcc -c90 --gnu`.

3.88.2 See also

- [--c90 on page 3-33](#)
- [--c99 on page 3-34](#)
- [--cpp on page 3-47](#)
- [--gnu_defaults on page 3-108](#)
- [--gnu_version=version on page 3-110](#)
- [--strict, --no_strict on page 3-194](#)
- [GNU extensions to the C and C++ languages on page 4-48](#)
- [Predefined macros on page 5-183](#).

3.89 `--gnu_defaults`

This option alters the default settings of certain other options to match the default behavior found in GCC. Platform-specific settings, such as those targeting ARM Linux, are unaffected.

3.89.1 Usage

`--gnu_defaults` does not imply specific targeting of ARM Linux.

When you use `--gnu_defaults`, the following options are enabled:

- `--allow_null_this`
- `--gnu`
- `--no_debug_macros`
- `--no_hide_all`
- `--no_implicit_include`
- `--signed_bitfields`
- `--wchar32`.

`--gnu` does not set these defaults. It only enables the GNU compiler extensions.

3.89.2 Default

When you use `--arm_linux` and other ARM Linux-targeting options, `--gnu_defaults` is automatically implied.

3.89.3 See also

- [--allow_null_this, --no_allow_null_this](#) on page 3-8
- [--arm_linux](#) on page 3-16
- [--debug_macros, --no_debug_macros](#) on page 3-57
- [--gnu](#) on page 3-107
- [--hide_all, --no_hide_all](#) on page 3-113
- [--implicit_include, --no_implicit_include](#) on page 3-116
- [--signed_bitfields, --unsigned_bitfields](#) on page 3-190
- [--wchar32](#) on page 3-225.

3.90 --gnu_instrument, --no_gnu_instrument

This option inserts GCC-style instrumentation calls for profiling entry and exit to functions.

3.90.1 Usage

After function entry and before function exit, the following profiling functions are called with the address of the current function and its call site:

```
void __cyg_profile_func_enter(void *current_func, void *callsite);  
void __cyg_profile_func_exit(void *current_func, void *callsite);
```

3.90.2 Restrictions

You must provide definitions of `__cyg_profile_func_enter()` and `__cyg_profile_func_exit()`.

It is necessary to explicitly mark `__cyg_profile_func_enter()` and `__cyg_profile_func_exit()` with `__attribute__((no_instrument_function))`.

3.90.3 See also

- [__attribute__\(\(no_instrument_function\)\) function attribute](#) on page 5-51.

3.91 `--gnu_version=version`

This option attempts to make the compiler compatible with a particular version of GCC.

3.91.1 Syntax

`--gnu_version=version`

Where *version* is a decimal number denoting the version of GCC that you are attempting to make the compiler compatible with.

———— **Note** —————

Currently, the maximum supported value for `--gnu_version` in `armcc v4.0` and later is `40400`, that is `gcc 4.4`. Although version numbers larger than this are permitted, `armcc` treats them the same as `40400`.

3.91.2 Mode

This option is for when GNU compatibility mode is being used.

3.91.3 Usage

This option is for expert use. It is provided for dealing with legacy code. You are not normally required to use it.

3.91.4 Default

In ARM Compiler 4.1 and later, the default is `40200`. This corresponds to GCC version 4.2.0.

3.91.5 Example

`--gnu_version=30401` makes the compiler compatible with GCC 3.4.1 as far as possible.

3.91.6 See also

- [--arm_linux_configure](#) on page 3-19
- [--gnu](#) on page 3-107.

3.92 `--guiding_decls`, `--no_guiding_decls`

This option enables or disables the recognition of guiding declarations for template functions in C++.

A *guiding declaration* is a function declaration that matches an instance of a function template but has no explicit definition because its definition derives from the function template.

If `--no_guiding_decls` is combined with `--old_specializations`, a specialization of a nonmember template function is not recognized. It is treated as a definition of an independent function.

Note

The option `--guiding_decls` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

3.92.1 Mode

This option is effective only if the source language is C++.

3.92.2 Default

The default is `--no_guiding_decls`.

3.92.3 Example

```
template <class T> void f(T)
{
    ...
}
void f(int);
```

When regarded as a guiding declaration, `f(int)` is an instance of the template. Otherwise, it is an independent function so you must supply a definition.

3.92.4 See also

- [--apcs=qualifer...qualifier](#) on page 3-11
- [--old_specializations](#), [--no_old_specializations](#) on page 3-158.

3.93 --help

This option displays a summary of the main command-line options.

3.93.1 Default

--help applies by default if you fail to specify any command-line options or source files.

3.93.2 See also

- [--show_cmdline](#) on page 3-189.
- [--vsn](#) on page 3-219

3.94 --hide_all, --no_hide_all

This option enables you to control symbol visibility when building SVr4 shared objects.

3.94.1 Usage

Use `--no_hide_all` to force the compiler to use `STV_DEFAULT` visibility for all extern variables and functions if they do not use `__declspec(dllexport)` or `__attribute__((visibility("visibility_type")))`. This also forces them to be pre-emptible at runtime by a dynamic loader.

When building a System V or ARM Linux shared library, use `--no_hide_all` together with `--apcs /fpic`.

Use `--hide_all` to set the visibility to `STV_HIDDEN`, so that symbols cannot be dynamically linked.

3.94.2 Default

The default is `--hide_all`.

3.94.3 See also

- [--apcs=qualifer...qualifier](#) on page 3-11
- [__attribute__\(\(visibility\("visibility_type"\)\)\)](#) function attribute on page 5-61
- [__attribute__\(\(visibility\("visibility_type"\)\)\)](#) variable attribute on page 5-81
- [__declspec\(dllexport\)](#) on page 5-31
- [__declspec\(dllimport\)](#) on page 5-33
- [--gnu_defaults](#) on page 3-108
- [Symbol visibility for BPABI models on page 10-7](#) in *Using the Linker*
- [--symver_script=filename](#) on page 2-167 in the *Linker Reference*
- [--visibility_inlines_hidden](#) on page 3-217.

3.95 `-I`*dir*[,*dir*,...]

This option adds the specified directory, or comma-separated list of directories, to the list of places that are searched to find included files.

If you specify more than one directory, the directories are searched in the same order as the `-I` options specifying them.

3.95.1 Syntax

`-I`*dir*[,*dir*,...]

Where:

dir[,*dir*,...] is a comma-separated list of directories to be searched for included files. At least one directory must be specified. When specifying multiple directories, do not include spaces between commas and directory names in the list.

3.95.2 See also

- [-J](#)*dir*[,*dir*,...] on page 3-125
- [--kandr_include](#) on page 3-126
- [--preinclude=filename](#) on page 3-172
- [--sys_include](#) on page 3-196
- [Compiler command-line options and search paths on page 3-19](#) in *Using the Compiler*
- [Factors influencing how the compiler searches for header files on page 3-18](#) in *Using the Compiler*.

3.96 `--ignore_missing_headers`

This option instructs the compiler to print dependency lines for header files even if the header files are missing. It only takes effect when dependency generation options (`--md` or `-M`) are specified.

Warning and error messages on missing header files are suppressed, and compilation continues.

3.96.1 Usage

This option is used for automatically updating makefiles. It is analogous to the GCC `-MG` command-line option.

3.96.2 See also

- [--depend=filename](#) on page 3-61
- [--depend_format=string](#) on page 3-63
- [--depend_system_headers, --no_depend_system_headers](#) on page 3-66
- [--depend_target=target](#) on page 3-67
- [-M](#) on page 3-144
- [--md](#) on page 3-145
- [--phony_targets](#) on page 3-170.

3.97 `--implicit_include`, `--no_implicit_include`

This option controls the implicit inclusion of source files as a method of finding definitions of template entities to be instantiated in C++.

3.97.1 Mode

This option is effective only if the source language is C++.

3.97.2 Default

The default is `--implicit_include`.

3.97.3 See also

- [--gnu_defaults](#) on page 3-108
- [--implicit_include_searches](#), [--no_implicit_include_searches](#) on page 3-117
- [Implicit inclusion](#) on page 6-19.

3.98 `--implicit_include_searches`, `--no_implicit_include_searches`

This option controls how the compiler searches for implicit include files for templates in C++.

When the option `--implicit_include_searches` is selected, the compiler uses the search path to look for implicit include files based on partial names of the form *filename.**. The search path is determined by `-I`, `-J`, the `ARMCCnINC` environment variable and the `ARMINC` environment variable. The search path also includes the default `./include` directory if `-J`, `ARMCCnINC`, and `ARMINC` are not set.

When the option `--no_implicit_include_searches` is selected, the compiler looks for implicit include files based on the full names of files, including path names.

3.98.1 Mode

This option is effective only if the source language is C++.

3.98.2 Default

The default is `--no_implicit_include_searches`.

3.98.3 See also

- [-Idir\[,dir,...\]](#) on page 3-114
- [--implicit_include, --no_implicit_include](#) on page 3-116
- [-Jdir\[,dir,...\]](#) on page 3-125
- [Implicit inclusion](#) on page 6-19
- [Compiler command-line options and search paths on page 3-19](#) in *Using the Compiler*
- [Toolchain environment variables on page 2-14](#) in *Introducing the ARM Compiler toolchain*.

3.99 `--implicit_key_function`, `--no_implicit_key_function`

These options control whether an implicitly instantiated template member function can be selected as a key function. (Normally the key, or decider, function for a class is its first non-inline virtual function, in declaration order, that is not pure virtual. However, in the case of an implicitly instantiated template function, the function would have vague linkage, that is, might be multiply defined.)

Remark #2819-D is produced when a key function is implicit. This remark can be seen with `--remarks` or with `--diag_warning=2819`.

3.99.1 Default

The default is `--implicit_key_function`.

3.99.2 See also

- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [--remarks](#) on page 3-181.

3.100 --implicit_typename, --no_implicit_typename

This option controls the implicit determination, from context, whether a template parameter dependent name is a type or nontype in C++.

———— **Note** —————

The option `--implicit_typename` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

3.100.1 Mode

This option is effective only if the source language is C++.

3.100.2 Default

The default is `--no_implicit_typename`.

———— **Note** —————

The `--implicit_typename` option has no effect unless you also specify `--no_parse_templates`.

3.100.3 See also

- [--dep_name, --no_dep_name](#) on page 3-60
- [--parse_templates, --no_parse_templates](#) on page 3-164
- [Template instantiation](#) on page 6-19.

3.101 `--import_all_vtbl`

This option causes external references to class impeditenta variables (vtables, RTTI, for example) to be marked as having dynamic linkage. It does not cause definitions of class impeditenta to have dynamic linkage.

3.101.1 See also

- [--export_all_vtbl, --no_export_all_vtbl](#) on page 3-90.

3.102 --info=totals

This option instructs the compiler to give totals of the object code and data size for each object file.

The compiler returns the same totals that `fromelf` returns when `fromelf --text -z` is used, in a similar format. The totals include embedded assembler sizes when embedded assembly exists in the source code.

3.102.1 Example

Code (inc. data)	RO Data	RW Data	ZI Data	Debug	File Name
3308	1556	0	44	10200	8402 dhry_1.o
Code (inc. data)	RO Data	RW Data	ZI Data	Debug	File Name
416	28	0	0	0	7722 dhry_2.o

The (inc. data) column gives the size of constants, string literals, and other data items used as part of the code. The Code column, shown in the example, *includes* this value.

3.102.2 See also

- [--list on page 3-133](#)
- [--info=topic\[,topic,...\] on page 2-80 in the Linker Reference](#)
- [--text on page 4-73 in Using the fromelf Image Converter](#)
- [Code metrics on page 6-15 in Using the Compiler.](#)

3.103 --inline, --no_inline

These options enable and disable the inlining of functions. Disabling the inlining of functions can help to improve the debug illusion.

When the option `--inline` is selected, the compiler considers inlining each function. Compiling your code with `--inline` does not guarantee that all functions are inlined. See [Compiler decisions on function inlining on page 6-31](#) in *Using the ARM Compiler* for more information about how the compiler decides to inline functions.

When the option `--no_inline` is selected, the compiler does not attempt to inline functions, other than functions qualified with `__forceinline`.

3.103.1 Default

The default is `--inline`.

3.103.2 See also

- [--autoinline, --no_autoinline on page 3-26](#)
- [--forceinline on page 3-95](#)
- [-Onum on page 3-156](#)
- [-Ospace on page 3-160](#)
- [-Otime on page 3-161](#)
- [__forceinline on page 5-10](#)
- [__inline on page 5-13](#)
- [Linker feedback during compilation on page 3-23](#) in *Using the Compiler*
- [Inline functions on page 6-30](#) in *Using the Compiler*.

3.104 --interface_enums_are_32_bit

This option helps to provide compatibility between external code interfaces, with regard to the size of enumerated types.

3.104.1 Usage

It is not possible to link an object file compiled with `--enum_is_int`, with another object file that is compiled without `--enum_is_int`. The linker is unable to determine whether or not the enumerated types are used in a way that affects the external interfaces, so on detecting these build differences, it produces a warning or an error. You can avoid this by compiling with `--interface_enums_are_32_bit`. The resulting object file can then be linked with any other object file, without the linker-detected conflict that arises from different enumeration type sizes.

———— **Note** —————

When you use this option, you are making a promise to the compiler that all the enumerated types used in your external interfaces are 32 bits wide. For example, if you ensure that every enum you declare includes at least one value larger than 2 to the power of 16, the compiler is forced to make the enum 32 bits wide, whether or not you use `--enum_is_int`. It is up to you to ensure that the promise you are making to the compiler is true. (Another method of satisfying this condition is to ensure that you have no enums in your external interface.)

3.104.2 Default

By default, the smallest data type that can hold the values of all enumerators is used.

3.104.3 See also

- [--enum_is_int](#) on page 3-85.

3.105 --interleave

This option interleaves C or C++ source code line by line as comments within an assembly listing generated using the `--asm` option or `-S` option.

3.105.1 Usage

The action of `--interleave` depends on the combination of options used:

Table 3-4 Compiling with the `---interleave` option

Compiler option	Action
<code>--asm --interleave</code>	Writes a listing to a file of the disassembly of the compiled source, interleaving the source code with the disassembly. The link step is also performed, unless the <code>-c</code> option is used. The disassembly is written to a text file whose name defaults to the name of the input file with the filename extension <code>.txt</code>
<code>-S --interleave</code>	Writes a listing to a file of the disassembly of the compiled source, interleaving the source code with the disassembly. The disassembly is written to a text file whose name defaults to the name of the input file with the filename extension <code>.txt</code>

3.105.2 Restrictions

- You cannot re-assemble an assembly listing generated with `--asm --interleave` or `-S --interleave`.
- Preprocessed source files contain `#line` directives. When compiling preprocessed files using `--asm --interleave` or `-S --interleave`, the compiler searches for the original files indicated by any `#line` directives, and uses the correct lines from those files. This ensures that compiling a preprocessed file gives exactly the same output and behavior as if the original files were compiled.

If the compiler cannot find the original files, it is unable to interleave the source. Therefore, if you have preprocessed source files with `#line` directives, but the original unpreprocessed files are not present, you must remove all the `#line` directives before you compile with `--interleave`.

3.105.3 See also

- [--asm on page 3-24](#)
- [-S on page 3-187](#).

3.106 `-Jdir[,dir,...]`

This option adds the specified directory, or comma-separated list of directories, to the list of system includes.

Downgradable errors, warnings, and remarks are suppressed, even if `--diag_error` is used.

Angle-bracketed include files are searched for first in the list of system includes, followed by any include list specified with the option `-I`.

———— **Note** —————

On Windows systems, you must enclose `ARMCCnINC` in double quotes if you specify this environment variable on the command line, because the default path defined by the variable contains spaces. For example:

```
armcc -J"%ARMCC5INC%" -c main.c
```

3.106.1 Syntax

`-Jdir[,dir,...]`

Where:

`dir[,dir,...]` is a comma-separated list of directories to be added to the list of system includes.

At least one directory must be specified.

When specifying multiple directories, do not include spaces between commas and directory names in the list.

3.106.2 See also

- [-Idir\[,dir,...\]](#) on page 3-114
- [--kandr_include](#) on page 3-126
- [--preinclude=filename](#) on page 3-172
- [--sys_include](#) on page 3-196
- [Factors influencing how the compiler searches for header files on page 3-18](#) in *Using the Compiler*
- [Compiler command-line options and search paths on page 3-19](#) in *Using the Compiler*
- [Toolchain environment variables on page 2-14](#) in *Introducing the ARM Compiler toolchain*.

3.107 --kandr_include

This option ensures that Kernighan and Ritchie search rules are used for locating included files. The current place is defined by the original source file and is not stacked.

3.107.1 Default

If you do not specify `--kandr_include`, Berkeley-style searching applies.

3.107.2 See also

- [-Idir\[,dir,...\]](#) on page 3-114
- [-Jdir\[,dir,...\]](#) on page 3-125
- [--preinclude=filename](#) on page 3-172
- [--sys_include](#) on page 3-196
- [Factors influencing how the compiler searches for header files on page 3-18](#) in *Using the Compiler*
- [Compiler search rules and the current place on page 3-20](#) in *Using the Compiler*.

3.108 -Lopt

This option specifies command-line options to pass to the linker when a link step is being performed after compilation. Options can be passed when creating a partially-linked object or an executable image.

3.108.1 Syntax

`-Lopt`

Where:

`opt` is a command-line option to pass to the linker.

3.108.2 Restrictions

If an unsupported Linker option is passed to it using `-L`, an error is generated by the linker.

3.108.3 Example

```
armcc main.c -L--map
```

3.108.4 See also

- [-Aopt](#) on page 3-6
- [--show_cmdline](#) on page 3-189.

3.109 --library_interface=*lib*

This option enables the generation of code that is compatible with the selected library type.

3.109.1 Syntax

```
--library_interface=lib
```

Where *lib* is one of:

none	Specifies that the compiler output works with any ISO C90 library, but does not use AEABI-defined library functions unless they are required for the code to behave correctly. For example, this option suppresses the use of AEABI-defined functions that are introduced only as an optimization such as <code>__aeabi_memcpy</code> .
armcc	Specifies that the compiler output works with the ARM runtime libraries in ARM Compiler 4.1 and later.
armcc_c90	Behaves similarly to <code>--library_interface=armcc</code> . The difference is that references in the input source code to function names that are not reserved by C90, are not modified by the compiler. Otherwise, some C99 <code>math.h</code> function names might be prefixed with <code>__hardfp_</code> , for example <code>__hardfp_tgamma</code> .
aeabi_clib90	Specifies that the compiler output works with any ISO C90 library compliant with the <i>ARM Embedded Application Binary Interface</i> (AEABI).
aeabi_clib99	Specifies that the compiler output works with any ISO C99 library compliant with the AEABI.
aeabi_clib	Specifies that the compiler output works with any ISO C library compliant with the AEABI. Selecting the option <code>--library_interface=aeabi_clib</code> is equivalent to specifying either <code>--library_interface=aeabi_clib90</code> or <code>--library_interface=aeabi_clib99</code> , depending on the choice of source language used. The choice of source language is dependent both on the command-line options selected and on the filename suffixes used.
aeabi_glibc	Specifies that the compiler output works with an AEABI-compliant version of the GNU C library.
aeabi_clib90_hardfp	Specifies that the compiler output works with any ISO C90 library compliant with the AEABI, and causes calls to the C library (including the math libraries) to call hardware floating-point library functions.
aeabi_clib99_hardfp	Specifies that the compiler output works with any ISO C99 library compliant with the AEABI, and causes calls to the C library (including the math libraries) to call hardware floating-point library functions.
aeabi_clib_hardfp	Specifies that the compiler output works with any ISO C library compliant with the AEABI. Selecting the option <code>--library_interface=aeabi_clib_hardfp</code> is equivalent to specifying either <code>--library_interface=aeabi_clib90_hardfp</code> or <code>--library_interface=aeabi_clib99_hardfp</code> , depending on the choice of source language used.

The choice of source language is dependent both on the command-line options selected and on the filename suffixes used.

Causes calls to the C library (including the math libraries) to call hardware floating-point library functions.

<code>aeabi_glibc_hardfp</code>	Specifies that the compiler output works with an AEABI-compliant version of the GNU C library, and causes calls to the C library (including the math libraries) to call hardware floating-point library functions.
<code>rvct30</code>	Specifies that the compiler output is compatible with RVCT 3.0 runtime libraries.
<code>rvct30_c90</code>	Behaves similarly to <code>rvct30</code> . In addition, specifies that the compiler output is compatible with any ISO C90 library.
<code>rvct31</code>	Specifies that the compiler output is compatible with RVCT 3.1 runtime libraries.
<code>rvct31_c90</code>	Behaves similarly to <code>rvct31</code> . In addition, specifies that the compiler output is compatible with any ISO C90 library.
<code>rvct40</code>	Specifies that the compiler output is compatible with RVCT 4.0 runtime libraries.
<code>rvct40_c90</code>	Behaves similarly to <code>rvct40</code> . In addition, specifies that the compiler output is compatible with any ISO C90 library.

3.109.2 Default

If you do not specify `--library_interface`, the compiler assumes `--library_interface=armcc`.

3.109.3 Usage

- Use the option `--library_interface=armcc` to exploit the full range of compiler and library optimizations when linking.
- Use an option of the form `--library_interface=aeabi_*` when linking with an ABI-compliant C library. Options of the form `--library_interface=aeabi_*` ensure that the compiler does not generate calls to any optimized functions provided by the ARM C library.
- It is an error to use any of the `_hardfp` library interfaces when compiling with `--fpu=softvfp`.

3.109.4 Example

If your code calls functions, provided by an embedded operating system, that replace functions provided by the ARM C library, then compile your code with the option `--library_interface=aeabi_clib`. This option disables calls to any special ARM variants of the library functions replaced by the operating system.

3.109.5 See also

- [Compliance with the Application Binary Interface \(ABI\) for the ARM architecture on page 2-9](#) in *Using ARM C and C++ Libraries and Floating-Point Support*.

3.110 --library_type=*lib*

This option enables the selected library to be used at link time.

———— **Note** —————

Use this option with the linker to override all other --library_type options.

3.110.1 Syntax

--library_type=*lib*

Where *lib* is one of:

standardlib Specifies that the full ARM Compiler 4.1 and later runtime libraries are selected at link time.

Use this option to exploit the full range of compiler optimizations when linking.

microlib Specifies that the C micro-library (microlib) is selected at link time.

3.110.2 Default

If you do not specify --library_type, the compiler assumes --library_type=standardlib.

3.110.3 See also

- [--library_type=lib on page 2-98](#) in the *Linker Reference*
- [About microlib on page 3-2](#) in *Using ARM C and C++ Libraries and Floating-Point Support*
- [Building an application with microlib on page 3-7](#) in *Using ARM C and C++ Libraries and Floating-Point Support*.

3.111 --licretry

If you are using floating licenses, this option makes up to 10 attempts to obtain a license when you invoke armcc.

3.111.1 Usage

Use this option if your builds are failing to obtain a license from your license server, and only after you have ruled out any other problems with the network or the license server setup.

It is recommended that you place this option in the ARMCCn_CCOPT environment variable. In this way, you do not have to modify your build files.

3.111.2 See also

- [--licretry on page 2-99](#) in the *Linker Reference*
- [--licretry on page 2-48](#) in the *Assembler Reference*
- [--licretry on page 4-51](#) in *Using the fromelf Image Converter*
- [Toolchain environment variables on page 2-14](#) in *Introducing ARM Compilation Tools*
- *ARM DS-5 License Management Guide*,
<http://infocenter.arm.com/help/topic/com.arm.doc.dui0577-/index.html>
- [Toolchain environment variables on page 2-14](#) in *Introducing the ARM Compiler toolchain*.

3.112 --link_all_input, --no_link_all_input

This option enables and disables the suppression of errors for unrecognized input filename extensions.

When enabled, the compiler suppresses errors for unrecognized input filename extensions, and treats all unrecognized input files as object files or libraries to be passed to the linker.

3.112.1 Default

The default is `--no_link_all_input`.

3.112.2 See also

- [--compile_all_input, --no_compile_all_input on page 3-37](#)
- [Filename suffixes recognized by the compiler on page 3-15](#) in *Using the Compiler*.

3.113 --list

This option instructs the compiler to generate raw listing information for a source file. The name of the raw listing file defaults to the name of the input file with the filename extension `.lst`.

If you specify multiple source files on the command line, the compiler generates listings for all of the source files, writing each to a separate listing file whose name is generated from the corresponding source file name. However, when `--multifile` is used, a concatenated listing is written to a single listing file, whose name is generated from the first source file name.

3.113.1 Usage

Typically, raw listing information is used to generate a formatted listing. The raw listing file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the compiler. Each line of the listing file begins with any of the following key characters that identifies the type of line:

- N A normal line of source. The rest of the line is the text of the line of source.
- X The expanded form of a normal line of source. The rest of the line is the text of the line. This line appears following the N line, and only if the line contains nontrivial modifications. Comments are considered trivial modifications, and macro expansions, line splices, and trigraphs are considered nontrivial modifications. Comments are replaced by a single space in the expanded-form line.
- S A line of source skipped by an `#if` or similar. The rest of the line is text.

———— **Note** —————

The `#else`, `#elseif`, or `#endif` that ends a skip is marked with an N.

- L Indicates a change in source position. That is, the line has a format similar to the `#` line-identifying directive output by the preprocessor:

`L line-number "filename" key`

where *key* can be:

- 1 For entry into an include file.
- 2 For exit from an include file.

Otherwise, *key* is omitted. The first line in the raw listing file is always an L line identifying the primary input file. L lines are also output for `#line` directives where *key* is omitted. L lines indicate the source position of the following source line in the raw listing file.

- R/W/E Indicates a diagnostic, where:

- R Indicates a remark.
- W Indicates a warning.
- E Indicates an error.

The line has the form:

`type "filename" line-number column-number message-text`

where *type* can be R, W, or E.

Errors at the end of file indicate the last line of the primary source file and a column number of zero.

Command-line errors are errors with a filename of "`<command line>`". No line or column number is displayed as part of the error message.

Internal errors are errors with position information as usual, and message-text beginning with (Internal fault).

When a diagnostic message displays a list, for example, all the contending routines when there is ambiguity on an overloaded call, the initial diagnostic line is followed by one or more lines with the same overall format. However, the code letter is the lowercase version of the code letter in the initial line. The source position in these lines is the same as that in the corresponding initial line.

3.113.2 Example

```
/* main.c */
#include <stdbool.h>
int main(void)
{
    return(true);
}
```

Compiling this code with the option `--list` produces the raw listing file:

```
L 1 "main.c"
N#include <stdbool.h>
L 1 "...include\...\stdbool.h" 1
N/* stdbool.h */
N
...
N #ifndef __cplusplus /* In C++, 'bool', 'true' and 'false' and keywords */
N #define bool _Bool
N #define true 1
N #define false 0
N #endif
...
L 2 "main.c" 2
N
Nint main(void)
N{
N return(true);
X return(1);
N}
```

3.113.3 See also

- [--asm](#) on page 3-24
- [-c](#) on page 3-31
- [--depend=filename](#) on page 3-61
- [--depend_format=string](#) on page 3-63
- [--info=totals](#) on page 3-121
- [--interleave](#) on page 3-124
- [--list_dir=directory_name](#) on page 3-135
- [--md](#) on page 3-145
- [-S](#) on page 3-187
- [Severity of compiler diagnostic messages on page 7-3](#) in *Using the Compiler*.

3.114 `--list_dir=directory_name`

This option enables you to specify a directory for `--list` output.

3.114.1 Example

```
armcc -c --list_dir=lst --list f1.c f2.c
```

Result:

```
lst/f1.lst  
lst/f2.lst
```

3.114.2 See also

- [--asm_dir=directory_name](#) on page 3-25
- [--depend_dir=directory_name](#) on page 3-62
- [--list](#) on page 3-133
- [--output_dir=directory_name](#) on page 3-162.

3.115 --list_macros

This option lists macro definitions to stdout after processing a specified source file. The listed output contains macro definitions that are used on the command line, predefined by the compiler, and found in header and source files, depending on usage.

3.115.1 Usage

To list macros that are defined on the command line, predefined by the compiler, and found in header and source files, use `--list_macros` with a non-empty source file.

To list only macros predefined by the compiler and specified on the command line, use `--list_macros` with an empty source file.

3.115.2 Restrictions

Code generation is suppressed.

3.115.3 See also

- [Predefined macros on page 5-183](#)
- [-Dname\[\(parm-list\)\]\[=def\] on page 3-54](#)
- [-E on page 3-82](#)
- [--show_cmdline on page 3-189](#)
- [--via=filename on page 3-216](#).

3.116 `--littleend`

This option instructs to the compiler to generate code for an ARM processor using little-endian memory.

With little-endian memory, the least significant byte of a word has the lowest address.

3.116.1 Default

The compiler assumes `--littleend` unless `--bigend` is explicitly specified.

3.116.2 See also

- [--bigend on page 3-27](#).

3.117 `--locale=lang_country`

This option switches the default locale for source files to the one you specify in *lang_country*.

3.117.1 Syntax

```
--locale=lang_country
```

Where:

lang_country is the new default locale.

Use this option in combination with `--multibyte_chars`.

3.117.2 Restrictions

The locale name might be case-sensitive, depending on the host platform.

The permitted settings of locale are determined by the host platform.

Ensure that you have installed the appropriate locale support for the host platform.

3.117.3 Example

To compile Japanese source files on an English-based Windows workstation, use:

```
--multibyte_chars --locale=japanese
```

and on a UNIX workstation use:

```
--multibyte_chars --locale=ja_JP
```

3.117.4 See also

- [--message_locale=lang_country\[.codepage\]](#) on page 3-146
- [--multibyte_chars](#), [--no_multibyte_chars](#) on page 3-149.

3.118 --long_long

This option permits use of the long long data type in strict mode.

3.118.1 Example

To successfully compile the following code in strict mode, you must use `--strict --long_long`.

```
long long f(long long x, long long y)
{
    return x*y;
}
```

3.118.2 See also

- [--strict, --no_strict](#) on page 3-194.

3.119 --loose_implicit_cast

This option makes illegal implicit casts legal, such as implicit casts of a nonzero integer to a pointer.

3.119.1 Example

```
int *p = 0x8000;
```

Compiling this example without the option `--loose_implicit_cast`, generates an error.

Compiling this example with the option `--loose_implicit_cast`, generates a warning message, that you can suppress.

3.120 --lower_ropi, --no_lower_ropi

This option enables or disables less restrictive C when compiling with --apcs=/ropi.

3.120.1 Default

The default is --no_lower_ropi.

———— **Note** —————

If you compile with --lower_ropi, then the static initialization is done at runtime by the C++ constructor mechanism for both C and C++ code. This enables these static initializations to work with ROPI code.

3.120.2 See also

- [--apcs=qualifer...qualifier](#) on page 3-11
- [--lower_rwpi, --no_lower_rwpi](#) on page 3-142
- [Code compatibility between separately compiled and assembled modules on page 3-22](#) in *Using the Compiler*.

3.121 --lower_rwpi, --no_lower_rwpi

This option enables or disables less restrictive C and C++ when compiling with --apcs=/rwpi.

3.121.1 Default

The default is --lower_rwpi.

———— **Note** —————

If you compile with --lower_rwpi, then the static initialization is done at runtime by the C++ constructor mechanism, even for C. This enables these static initializations to work with RWPI code.

3.121.2 See also

- [--apcs=qualifer...qualifier](#) on page 3-11
- [--lower_ropi, --no_lower_ropi](#) on page 3-141
- [Code compatibility between separately compiled and assembled modules on page 3-22](#) in *Using the Compiler*.

3.122 `--ltcg`

This option instructs the compiler to create objects in an intermediate format so that *Link-Time Code Generation* (LTCG) optimizations can be performed. The optimizations applied include cross-module inlining to improve performance, and sharing of base addresses to reduce code size.

Note

- This option might significantly increase link time and memory requirements. For large applications it is recommended that you do the code generation in partial link steps with a subset of the objects.
 - The LTCG feature is deprecated. As an alternative ARM recommends you use the `--multifile` option.
-

3.122.1 Example

The following example shows how to use the `--ltcg` option.

```
armcc -c --ltcg file1.c
armcc -c --ltcg file2.c
armlink --ltcg file1.o file2.o -o prog.axf
```

3.122.2 See also

- [--multifile, --no_multifile](#) on page 3-150
- [-Onum](#) on page 3-156
- [Automatic function inlining and multifile compilation on page 6-35](#) in Using the Compiler
- [Inline functions in C99 mode on page 6-39](#) in Using the Compiler
- [--ltcg](#) on page 2-106 in the *Linker Reference*
- [About link-time code generation on page 5-11](#) in the *Linker Reference*.

3.123 -M

This option instructs the compiler to produce a list of makefile dependency lines suitable for use by a make utility.

The compiler executes only the preprocessor step of the compilation. By default, output is on the standard output stream.

If you specify multiple source files, a single dependency file is created.

If you specify the `-o filename` option, the dependency lines generated on standard output make reference to `filename.o`, and not to `source.o`. However, no object file is produced with the combination of `-M -o filename`.

Use the `--md` option to generate dependency lines and object files for each source file.

3.123.1 Example

You can redirect output to a file by using standard UNIX and MS-DOS notation, for example:

```
armcc -M source.c > Makefile
```

3.123.2 See also

- [-C on page 3-32](#)
- [--depend=filename on page 3-61](#)
- [--depend_system_headers, --no_depend_system_headers on page 3-66](#)
- [-E on page 3-82](#)
- [--md on page 3-145](#)
- [--depend_single_line, --no_depend_single_line on page 3-65](#)
- [-o filename on page 3-154](#).

3.124 --md

This option instructs the compiler to compile the source and write makefile dependency lines to a file.

The output file is suitable for use by a make utility.

The compiler names the file *filename.d*, where *filename* is the name of the source file. If you specify multiple source files, a dependency file is created for each source file.

If you want to produce makefile dependencies and preprocessor source file output in a single step, you can do so using the combination `--md -E` (or `--md -P` to suppress line number generation).

3.124.1 See also

- [--depend=filename](#) on page 3-61
- [--depend_format=string](#) on page 3-63
- [--depend_system_headers, --no_depend_system_headers](#) on page 3-66
- [-E](#) on page 3-82
- [-M](#) on page 3-144
- [--depend_single_line, --no_depend_single_line](#) on page 3-65
- [-o filename](#) on page 3-154.

3.125 `--message_locale=lang_country[.codepage]`

This option switches the default language for the display of error and warning messages to the one you specify in `lang_country` or `lang_country.codepage`.

3.125.1 Syntax

```
--message_locale=lang_country[.codepage]
```

Where:

`lang_country[.codepage]`

is the new default language for the display of error and warning messages.

The permitted languages are independent of the host platform.

The following settings are supported:

- `en_US`
- `zh_CN`
- `ko_KR`
- `ja_JP`.

3.125.2 Default

If you do not specify `--message_locale`, the compiler assumes `--message_locale=en_US`.

3.125.3 Restrictions

Ensure that you have installed the appropriate locale support for the host platform.

The locale name might be case-sensitive, depending on the host platform.

The ability to specify a codepage, and its meaning, depends on the host platform.

3.125.4 Errors

If you specify a setting that is not supported, the compiler generates an error message.

3.125.5 Example

To display messages in Japanese, use:

```
--message_locale=ja_JP
```

3.125.6 See also

- [--locale=lang_country](#) on page 3-138
- [--multibyte_chars, --no_multibyte_chars](#) on page 3-149.

3.126 --min_array_alignment=*opt*

This option enables you to specify the minimum alignment of arrays.

3.126.1 Syntax

```
--min_array_alignment=opt
```

Where:

<i>opt</i>	specifies the minimum alignment of arrays. The value of <i>opt</i> is:
1	byte alignment, or unaligned
2	two-byte, halfword alignment
4	four-byte, word alignment
8	eight-byte, doubleword alignment.

3.126.2 Usage

Use of this option is not recommended, unless required in certain specialized cases. For example, porting code to systems that have different data alignment requirements. Use of this option can result in increased code size at the higher *opt* values, and reduced performance at the lower *opt* values. If you only want to affect the alignment of specific arrays (rather than all arrays), use the `__align` keyword instead.

3.126.3 Default

If you do not use this option, arrays are unaligned (byte aligned).

3.126.4 Example

Compiling the following code with `--min_array_alignment=8` gives the alignment described in the comments:

```
char arr_c1[1];    // alignment == 8
char c1;          // alignment == 1
```

3.126.5 See also

- [__align](#) on page 5-6
- [__ALIGNOF__](#) on page 5-8.

3.127 --mm

This option has the same effect as `-M --no_depend_system_headers`.

3.127.1 See also

- [--depend_system_headers, --no_depend_system_headers](#) on page 3-66
- [-M](#) on page 3-144.

3.128 --multibyte_chars, --no_multibyte_chars

This option enables or disables processing for multibyte character sequences in comments, string literals, and character constants.

3.128.1 Default

The default depends on the regional language settings of your machine. For example:

- `--no_multibyte_chars` is the default for English
- `--multibyte_chars` is the default for Japanese.

3.128.2 Usage

Multibyte encodings are used for character sets such as the Japanese *Shift-Japanese Industrial Standard* (Shift-JIS).

3.128.3 See also

- [--locale=lang_country](#) on page 3-138
- [--message_locale=lang_country\[.codepage\]](#) on page 3-146.

3.129 --multifile, --no_multifile

This option enables or disables multifile compilation.

When `--multifile` is selected, the compiler performs optimizations across all files specified on the command line, instead of on each individual file. The specified files are compiled into one single object file.

The combined object file is named after the first source file you specify on the command line. To specify a different name for the combined object file, use the `-o filename` option.

An empty object file is created for each subsequent source file specified on the command line to meet the requirements of standard make systems.

———— **Note** —————

Compiling with `--multifile` has no effect if only a single source file is specified on the command line.

3.129.1 Default

The default is `--no_multifile`.

3.129.2 Usage

When `--multifile` is selected, the compiler might be able to perform additional optimizations by compiling across several source files.

There is no limit to the number of source files that can be specified on the command line, but ten files is a practical limit, because `--multifile` requires large amounts of memory while compiling. For the best optimization results, choose small groups of functionally related source files.

3.129.3 Example

```
armcc -c --multifile test1.c ... testn.c -o test.o
```

The resulting object file is named `test.o`, instead of `test1.c`, and empty object files `test2.o` to `testn.o` are created for each source file `test1.c ... testn.c` specified on the command line.

3.129.4 See also

- [-c on page 3-31](#)
- [--default_extension=ext on page 3-59](#)
- [--ltcg on page 3-143](#)
- [-o filename on page 3-154](#)
- [-Onum on page 3-156](#)
- [--whole_program on page 3-226](#)
- [Predefined macros on page 5-183.](#)

3.130 --multiply_latency=cycles

This option tells the compiler the number of cycles used by the hardware multiplier.

3.130.1 Syntax

```
--multiply_latency=cycles
```

Where *cycles* is the number of cycles used.

3.130.2 Usage

Use this option to tell the compiler how many cycles the MUL instruction takes to use the multiplier block and related parts of the chip. Until finished, these parts of the chip cannot be used for another instruction and the result of the MUL is not available for any later instructions to use.

It is possible that a processor might have two or more multiplier options that are set for a given hardware implementation. For example, one implementation might be configured to take one cycle to execute. The other implementation might take 33 cycles to execute. This option is used to convey the correct number of cycles for a given processor.

3.130.3 Default

The default number of cycles used by the hardware multiplier is processor-specific. See the Technical Reference Manual for the processor architecture you are compiling for.

3.130.4 Example

```
--multiply_latency=33
```

3.130.5 See also

- *Cortex-M1 Technical Reference Manual.*

3.131 --narrow_volatile_bitfields

The AEABI specifies that volatile bitfields are accessed as the size of their container type. However, some versions of GCC instead use the smallest access size that contains the entire bitfield. `--narrow_volatile_bitfields` emulates this non-AEABI compliant behavior.

3.131.1 See also

- *Application Binary Interface (ABI) for the ARM Architecture*, <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.abi/index.html>

3.132 `--nonstd_qualifier_deduction`, `--no_nonstd_qualifier_deduction`

This option controls whether or not nonstandard template argument deduction is to be performed in the qualifier portion of a qualified name in C++.

With this feature enabled, a template argument for the template parameter `T` can be deduced in contexts like `A<T>::B` or `T::B`. The standard deduction mechanism treats these as nondeduced contexts that use the values of template parameters that were either explicitly specified or deduced elsewhere.

———— **Note** —————

The option `--nonstd_qualifier_deduction` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

3.132.1 Mode

This option is effective only if the source language is C++.

3.132.2 Default

The default is `--no_nonstd_qualifier_deduction`.

3.133 -o filename

This option specifies the name of the output file. The full name of the output file produced depends on the combination of options used, as described in [Table 3-5](#) and [Table 3-6](#).

3.133.1 Syntax

If you specify a -o option, the compiler names the output file according to the conventions of [Table 3-5](#).

Table 3-5 Compiling with the -o option

Compiler option	Action	Usage notes
-o-	writes output to the standard output stream	<i>filename</i> is -. -S is assumed unless -E is specified.
-o <i>filename</i>	produces an executable image with name <i>filename</i>	
-c -o <i>filename</i>	produces an object file with name <i>filename</i>	
-S -o <i>filename</i>	produces an assembly language file with name <i>filename</i>	
-E -o <i>filename</i>	produces a file containing preprocessor output with name <i>filename</i>	

———— **Note** —————

This option overrides the --default_extension option.

3.133.2 Default

If you do not specify a -o option, the compiler names the output file according to the conventions of [Table 3-6](#).

Table 3-6 Compiling without the -o option

Compiler option	Action	Usage notes
-c	produces an object file whose name defaults to the name of the input file with the filename extension .o	
-S	produces an output file whose name defaults to the name of the input file with the filename extension .s	
-E	writes output from the preprocessor to the standard output stream	
(No option)	produces an executable image with the default name of __image.axf	none of -o, -c, -E or -S is specified on the command line

3.133.3 See also

- [--asm](#) on page 3-24

- `-c` on page 3-31
- `--default_extension=ext` on page 3-59
- `--depend=filename` on page 3-61
- `--depend_format=string` on page 3-63
- `-E` on page 3-82
- `--interleave` on page 3-124
- `--list` on page 3-133
- `--md` on page 3-145
- `-S` on page 3-187.

3.134 -O*num*

This option specifies the level of optimization to be used when compiling source files.

3.134.1 Syntax

-O*num*

Where *num* is one of the following:

- | | |
|---|---|
| 0 | Minimum optimization. Turns off most optimizations. It gives the best possible debug view and the lowest level of optimization. |
| 1 | Restricted optimization. Removes unused inline functions and unused static functions. Turns off optimizations that seriously degrade the debug view. If used with <code>--debug</code> , this option gives a satisfactory debug view with good code density. |
| 2 | High optimization. If used with <code>--debug</code> , the debug view might be less satisfactory because the mapping of object code to source code is not always clear. This is the default optimization level. |
| 3 | Maximum optimization. -O3 performs the same optimizations as -O2 however the balance between space and time optimizations in the generated code is more heavily weighted towards space or time compared with -O2. That is: <ul style="list-style-type: none"> • -O3 -Otime aims to produce faster code than -O2 -Otime, at the risk of increasing your image size • -O3 -Ospace aims to produce smaller code than -O2 -Ospace, but performance might be degraded. |

In addition, -O3 performs extra optimizations that are more aggressive, such as:

- High-level scalar optimizations, including loop unrolling, for -O3 -Otime. This can give significant performance benefits at a small code size cost, but at the risk of a longer build time.
- More aggressive inlining and automatic inlining for -O3 -Otime.

Note

The performance of floating-point code can be influenced by selecting an appropriate numerical model using the `--fpmode` option.

Note

Do not rely on the implementation details of these optimizations, because they might change in future releases.

3.134.2 Default

If you do not specify -O*num*, the compiler assumes -O2.

3.134.3 See also

- [--autoinline, --no_autoinline](#) on page 3-26
- [--debug, --no_debug](#) on page 3-56
- [--forceinline](#) on page 3-95
- [--fpmode=model](#) on page 3-97
- [--inline, --no_inline](#) on page 3-122

- [--lto on page 3-143](#)
- [--multifile, --no_multifile on page 3-150](#)
- [-Ospace on page 3-160](#)
- [-Otime on page 3-161](#)
- [The compiler as an optimizing compiler on page 6-5 in Using the Compiler.](#)

3.135 `--old_specializations`, `--no_old_specializations`

This option controls the acceptance of old-style template specializations in C++.

Old-style template specializations do not use the `template<>` syntax.

———— **Note** —————

The option `--old_specializations` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

3.135.1 Mode

This option is effective only if the source language is C++.

3.135.2 Default

The default is `--no_old_specializations`.

3.136 --old_style_preprocessing

This option performs preprocessing in the style of legacy compilers that do not follow the ISO C Standard.

3.136.1 See also

- [-E on page 3-82.](#)

3.137 -Ospace

This option instructs the compiler to perform optimizations to reduce image size at the expense of a possible increase in execution time.

Use this option if code size is more critical than performance. For example, when the `-Ospace` option is selected, large structure copies are done by out-of-line function calls instead of inline code.

If required, you can compile the time-critical parts of your code with `-Otime`, and the rest with `-Ospace`.

3.137.1 Default

If you do not specify either `-Ospace` or `-Otime`, the compiler assumes `-Ospace`.

3.137.2 See also

- [-Otime on page 3-161](#)
- [-Onum on page 3-156](#)
- [#pragma Onum on page 5-103](#)
- [#pragma Ospace on page 5-105](#)
- [#pragma Otime on page 5-106.](#)

3.138 -Otime

This option instructs the compiler to perform optimizations to reduce execution time at the expense of a possible increase in image size.

Use this option if execution time is more critical than code size. If required, you can compile the time-critical parts of your code with `-Otime`, and the rest with `-Ospace`.

3.138.1 Default

If you do not specify `-Otime`, the compiler assumes `-Ospace`.

3.138.2 Example

When the `-Otime` option is selected, the compiler compiles:

```
while (expression) body;
```

as:

```
if (expression)  
{  
    do body;  
    while (expression);  
}
```

3.138.3 See also

- [--multifile, --no_multifile](#) on page 3-150
- [-Onum](#) on page 3-156
- [-Ospace](#) on page 3-160
- [#pragma Onum](#) on page 5-103
- [#pragma Ospace](#) on page 5-105
- [#pragma Otime](#) on page 5-106.

3.139 `--output_dir=directory_name`

This option enables you to specify an output directory for object files and depending on the other options you use, certain other types of compiler output.

The directory for assembler output can be specified using `--asm_dir`. The directory for dependency output can be specified using `--depend_dir`. The directory for `--list` output can be specified using `--list_dir`. If these options are not used, the corresponding output is placed in the directory specified by `--output_dir`, or if `--output_dir` is not specified, in the default location (for example, the current directory).

The executable is placed in the default location.

3.139.1 Example

```
armcc -c --output_dir=obj f1.c f2.c
```

Result:

```
obj/f1.o  
obj/f2.o
```

3.139.2 See also

- [--asm_dir=directory_name](#) on page 3-25
- [--depend_dir=directory_name](#) on page 3-62
- [--list_dir=directory_name](#) on page 3-135.

3.140 -P

This option preprocesses source code without compiling, but does not generate line markers in the preprocessed output.

3.140.1 Usage

This option can be of use when the preprocessed output is destined to be parsed by a separate script or utility.

3.140.2 See also

- [-E on page 3-82.](#)

3.141 --parse_templates, --no_parse_templates

This option enables or disables the parsing of nonclass templates in their generic form in C++, that is, when the template is defined and before it is instantiated.

———— **Note** —————

The option `--no_parse_templates` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

3.141.1 Mode

This option is effective only if the source language is C++.

3.141.2 Default

The default is `--parse_templates`.

———— **Note** —————

`--no_parse_templates` cannot be used with `--dep_name`, because parsing is done by default if dependent name processing is enabled. Combining these options generates an error.

3.141.3 See also

- [--dep_name, --no_dep_name](#) on page 3-60
- [Template instantiation](#) on page 6-19.

3.142 --pch

This option instructs the compiler to use a PCH file if it exists, and to create a PCH file otherwise.

When the option `--pch` is specified, the compiler searches for a PCH file with the name *filename.pch*, where *filename.** is the name of the primary source file. The compiler uses the PCH file *filename.pch* if it exists, and creates a PCH file named *filename.pch* in the same directory as the primary source file otherwise.

3.142.1 Restrictions

This option has no effect if you include either the option `--use_pch=filename` or the option `--create_pch=filename` on the same command line.

3.142.2 See also

- [--create_pch=filename](#) on page 3-53
- [--pch_dir=dir](#) on page 3-166
- [--pch_messages, --no_pch_messages](#) on page 3-167
- [--pch_verbose, --no_pch_verbose](#) on page 3-168
- [--use_pch=filename](#) on page 3-211
- [#pragma hdrstop](#) on page 5-97
- [#pragma no_pch](#) on page 5-102
- [PreCompiled Header \(PCH\) files](#) on page 5-33 in *Using the Compiler*.

3.143 --pch_dir=*dir*

This option enables you to specify the directory where PCH files are stored. The directory is accessed whenever PCH files are created or used.

You can use this option with automatic or manual PCH mode.

3.143.1 Syntax

--pch_dir=*dir*

Where:

dir is the name of the directory where PCH files are stored.

If *dir* is unspecified, the compiler faults use of --pch_dir.

3.143.2 Errors

If the specified directory *dir* does not exist, the compiler generates an error.

3.143.3 See also

- [--create_pch=filename](#) on page 3-53
- [--pch](#) on page 3-165
- [--pch_messages, --no_pch_messages](#) on page 3-167
- [--pch_verbose, --no_pch_verbose](#) on page 3-168
- [--use_pch=filename](#) on page 3-211
- [#pragma hdrstop](#) on page 5-97
- [#pragma no_pch](#) on page 5-102
- [PreCompiled Header \(PCH\) files on page 5-33](#) in *Using the Compiler*.

3.144 --pch_messages, --no_pch_messages

This option enables or disables the display of messages indicating that a PCH file is used in the current compilation.

3.144.1 Default

The default is `--pch_messages`.

3.144.2 See also

- [--create_pch=filename](#) on page 3-53
- [--pch](#) on page 3-165
- [--pch_dir=dir](#) on page 3-166
- [--pch_verbose, --no_pch_verbose](#) on page 3-168
- [--use_pch=filename](#) on page 3-211
- [#pragma hdrstop](#) on page 5-97
- [#pragma no_pch](#) on page 5-102
- [PreCompiled Header \(PCH\) files](#) on page 5-33 in *Using the Compiler*.

3.145 --pch_verbose, --no_pch_verbose

This option enables or disables the display of messages giving reasons why a file cannot be precompiled.

In automatic PCH mode, this option ensures that for each PCH file that cannot be used for the current compilation, a message is displayed giving the reason why the file cannot be used.

3.145.1 Default

The default is `--no_pch_verbose`.

3.145.2 See also

- [--create_pch=filename](#) on page 3-53
- [--pch](#) on page 3-165
- [--pch_dir=dir](#) on page 3-166
- [--pch_messages, --no_pch_messages](#) on page 3-167
- [--use_pch=filename](#) on page 3-211
- [#pragma hdrstop](#) on page 5-97
- [#pragma no_pch](#) on page 5-102
- [PreCompiled Header \(PCH\) files on page 5-33](#) in *Using the Compiler*.

3.146 `--pending_instantiations=n`

This option specifies the maximum number of concurrent instantiations of a template in C++.

3.146.1 Syntax

`--pending_instantiations=n`

Where:

n is the maximum number of concurrent instantiations permitted.

If n is zero, there is no limit.

3.146.2 Mode

This option is effective only if the source language is C++.

3.146.3 Default

If you do not specify a `--pending_instantiations` option, then the compiler assumes `--pending_instantiations=64`.

3.146.4 Usage

Use this option to detect runaway recursive instantiations.

3.147 --phony_targets

This option instructs the compiler to emit dummy makefile rules. These rules work around make errors that are generated if you remove header files without a corresponding update to the makefile.

This option is analogous to the GCC command-line option, `-MP`.

3.147.1 Example

Example output:

```
source.o: source.c
source.o: header.h
header.h:
```

3.147.2 See also

- [--depend=filename](#) on page 3-61
- [--depend_format=string](#) on page 3-63
- [--depend_system_headers, --no_depend_system_headers](#) on page 3-66
- [--depend_target=target](#) on page 3-67
- [--ignore_missing_headers](#) on page 3-115
- [-M](#) on page 3-144
- [--md](#) on page 3-145

3.148 `--pointer_alignment=num`

This option specifies the unaligned pointer support required for an application.

3.148.1 Syntax

`--pointer_alignment=num`

Where *num* is one of:

- | | |
|---|--|
| 1 | Treats accesses through pointers as having an alignment of one, that is, byte-aligned or unaligned. |
| 2 | Treats accesses through pointers as having an alignment of at most two, that is, at most halfword aligned. |
| 4 | Treats accesses through pointers as having an alignment of at most four, that is, at most word aligned. |
| 8 | Accesses through pointers have normal alignment, that is, at most doubleword aligned. |

If *num* is unspecified, the compiler faults use of `--pointer_alignment`.

3.148.2 Usage

This option can help you port source code that has been written for architectures without alignment requirements. You can achieve finer control of access to unaligned data, with less impact on the quality of generated code, using the `__packed` qualifier.

3.148.3 Restrictions

De-aligning pointers might increase the code size, even on CPUs with unaligned access support. This is because only a subset of the load and store instructions benefit from unaligned access support. The compiler is unable to use multiple-word transfers or coprocessor-memory transfers, including hardware floating-point loads and stores, directly on unaligned memory objects.

———— **Note** —————

- Code size might increase significantly when compiling for CPUs without hardware support for unaligned access, for example, pre-v6 architectures.
- This option does not affect the placement of objects in memory, nor the layout and padding of structures.

3.148.4 See also

- [__packed](#) on page 5-17
- [#pragma pack\(n\)](#) on page 5-107
- [Compiler storage of data objects by natural byte alignment](#) on page 6-44 in *Using the Compiler*.

3.149 --preinclude=*filename*

This option instructs the compiler to include the source code of the specified file at the beginning of the compilation.

3.149.1 Syntax

```
--preinclude=filename
```

Where:

filename is the name of the file whose source code is to be included.

If *filename* is unspecified, the compiler faults use of --preinclude.

3.149.2 Usage

Use this option to establish standard macro definitions. The *filename* is searched for in the directories on the include search list.

It is possible to repeatedly specify this option on the command line. This results in pre-including the files in the order specified.

3.149.3 Restrictions

Sub-directories of directories specified on the include search list are not searched unless you use --arm_linux. If you use --arm_linux, the compiler includes the arm_linux subdirectory in its search for pre-include files.

3.149.4 Example

```
armcc --preinclude file1.h --preinclude file2.h -c source.c
```

3.149.5 See also

- [-I*dir*\[,*dir*,...\]](#) on page 3-114
- [-J*dir*\[,*dir*,...\]](#) on page 3-125
- [--kandr_include](#) on page 3-126
- [--sys_include](#) on page 3-196
- [Factors influencing how the compiler searches for header files on page 3-18](#) in *Using the Compiler*.

3.150 --preprocess_assembly

This option relaxes certain rules when producing preprocessed compiler output, to provide greater flexibility when preprocessing assembly language source code.

3.150.1 Usage

Use this option to relax certain preprocessor rules when generating preprocessed output from assembly language source files. Specifically, the following special cases are permitted that would normally produce a compiler error:

- Lines beginning with a '#' character followed by a space and a number, that would normally indicate a GNU non-standard line marker, are ignored and copied verbatim into the preprocessed output.
- Unrecognized preprocessing directives are ignored and copied verbatim into the preprocessed output.
- Where the token-paste '#' operator is used in a function-like macro, if it is used with a name that is not a macro parameter, the name is copied verbatim into the preprocessed output together with the preceding '#' character.

For example if the source file contains:

```
# define mymacro(arg) foo #bar arg
mymacro(x)
```

using the --preprocess_assembly option produces a preprocessed output that contains:

```
foo #bar x
```

3.150.2 Restrictions

This option is only valid when producing preprocessed output without continuing compilation, for example when using the -E, -P or -C command line options. It is ignored in other cases.

3.150.3 See also

- [-C on page 3-32](#)
- [-E on page 3-82](#)
- [-P on page 3-163](#).

3.151 --preprocessed

This option forces the preprocessor to handle files with `.i` filename extensions as if macros have already been substituted.

3.151.1 Usage

This option gives you the opportunity to use a different preprocessor. Generate your preprocessed code and then give the preprocessed code to the compiler in the form of a *filename.i* file, using `--preprocessed` to inform the compiler that the file has already been preprocessed.

3.151.2 Restrictions

This option only applies to macros. Trigraphs, line concatenation, comments and all other preprocessor items are preprocessed by the preprocessor in the normal way.

If you use `--compile_all_input`, the `.i` file is treated as a `.c` file. The preprocessor behaves as if no prior preprocessing has occurred.

3.151.3 Example

```
armcc --preprocessed foo.i -c -o foo.o
```

3.151.4 See also

- [--compile_all_input, --no_compile_all_input on page 3-37](#)
- [-E on page 3-82](#).

3.152 --project=*filename*, --no_project

The option `--project=filename` instructs the compiler to load the project template file specified by *filename*.

———— **Note** —————

To use *filename* as a default project file, set the RVDS_PROJECT environment variable to *filename*.

The option `--no_project` prevents the default project template file specified by the environment variable RVDS_PROJECT from being used.

———— **Note** —————

This option is deprecated.

3.152.1 Syntax

`--project=filename`

`--no_project`

Where:

filename is the name of a project template file.

If *filename* is unspecified, the compiler faults use of `--project`.

3.152.2 Restrictions

Options from a project template file are only set when they do not conflict with options already set on the command line. If an option from a project template file conflicts with an existing command-line option, the command-line option takes precedence.

3.152.3 Example

Consider the following project template file:

```
<!-- suiteconf.cfg -->
<suiteconf name="Platform Baseboard for ARM926EJ-S">
  <tool name="armcc">
    <cmdline>
      --cpu=ARM926EJ-S
      --fpu=vfpv2
    </cmdline>
  </tool>
</suiteconf>
```

When the RVDS_PROJECT environment variable is set to point to this file, the command:

```
armcc -c foo.c
```

results in an actual command line of:

```
armcc --cpu=ARM926EJ-S --fpu=VFPv2 -c foo.c
```

3.152.4 See also

- [--reinitialize_workdir](#) on page 3-179
- [--workdir=directory](#) on page 3-227.

3.153 --protect_stack, --no_protect_stack

The option `--protect_stack` inserts a guard variable onto the stack frame for each vulnerable function. The guard variable is inserted between any buffers and the return address entry.

A function is considered vulnerable if it contains a vulnerable array. A vulnerable array is one that:

- has automatic storage duration
- has a character type (`char` or `wchar_t`).

In addition to inserting the guard variable and check, the compiler also moves vulnerable arrays to the top of the stack, immediately preceding the guard variable. A copy of the guard variable's value is stored at another location, and is used to check that the guard has not been overwritten, indicating a buffer overflow.

3.153.1 Usage

Use `--protect_stack` to enable the stack protection feature. Use `--no_protect_stack` to explicitly disable this feature. If both options are specified, the last option specified takes effect.

The `--protect_stack_all` option adds this protection to all functions regardless of their vulnerability.

With stack protection, when a vulnerable function is called, the initial value of its guard variable is taken from a global variable:

```
void *__stack_chk_guard;
```

You must provide this variable with a suitable value, such as a random value. The value can change during the life of the program. For example, a suitable implementation might be to have the value constantly changed by another thread. In addition, you must implement this function:

```
void __stack_chk_fail(void);
```

It is called by the checking code on detection of corruption of the guard. In general, such a function would exit, possibly after reporting a fault.

For consistency with GNU tools, the option `-fstack-protector` is treated identically to `--protect-stack`. Similarly, the `-fstack-protector-all` option is treated identically to `--protect_stack_all`.

3.153.2 Example

In the following function, the array `buf` is vulnerable and the function is protected when compiled with `--protect-stack`:

```
void copy(const char *p)
{
    char buf[4];
    strcpy(buf, p);
}
```

3.153.3 Default

The default is `--no_protect_stack`.

3.154 --reassociate_saturation, --no_reassociate_saturation

These options enable and disable more aggressive optimization in loops that use saturating arithmetic.

3.154.1 Usage

Saturating additions do not have associative property. Therefore by default, the compiler does not re-associate saturating additions. `--reassociate_saturation` instructs the compiler to re-associate saturating additions to perform optimization. `--no_reassociate_saturation` prohibits re-association of saturating addition, and thus limits the level of optimization on saturating arithmetic. `--reassociate_saturation` enables optimizations such as vectorizing code when automatic vectorization is enabled using `--vectorize`. It also enables other optimizations when `--vectorize` is not specified, for example when compiling with `-O3 -Otime`.

3.154.2 Restriction

Saturating addition is not associative, so enabling `--reassociate_saturation` could affect the result with a reduction in accuracy.

3.154.3 Default

The default is `--no_reassociate_saturation`.

3.154.4 Example

The following code contains the function `L_mac`, which performs saturating additions. Therefore the compiler does not vectorize this code unless `--reassociate_saturation` and `--vectorize` are specified.

```
#include <dspfns.h>
int f(short *a, short *b)
{
    int i;
    int r = 0;
    for (i = 0; i < 100; i++)
        r=L_mac(r,a[i],b[i]);
    return r;
}
```

3.154.5 See also

- [--vectorize, --no_vectorize on page 3-213](#)
- [Automatic vectorization on page 4-9](#) in *Using the Compiler*.

3.155 --reduce_paths, --no_reduce_paths

This option enables or disables the elimination of redundant path name information in file paths.

When elimination of redundant path name information is enabled, the compiler removes sequences of the form `xyz\..` from directory paths passed to the operating system. This includes system paths constructed by the compiler itself, for example, for `#include` searching.

———— **Note** —————

The removal of sequences of the form `xyz\..` might not be valid if `xyz` is a link.

3.155.1 Mode

This option is effective on Windows systems only.

3.155.2 Usage

Windows systems impose a 260 character limit on file paths. Where path names exist whose absolute names expand to longer than 260 characters, you can use the `--reduce_paths` option to reduce absolute path name length by matching up directories with corresponding instances of `..` and eliminating the `directory/..` sequences in pairs.

———— **Note** —————

It is recommended that you avoid using long and deeply nested file paths, in preference to minimizing path lengths using the `--reduce_paths` option.

3.155.3 Default

The default is `--no_reduce_paths`.

3.155.4 Example

Compiling the file

```
..\..\..\xyzy\xyzy\objects\file.c
```

from the directory

```
\foo\bar\baz\gazonk\quux\bop
```

results in an actual path of

```
\foo\bar\baz\gazonk\quux\bop\..\..\..\xyzy\xyzy\objects\file.o
```

Compiling the same file from the same directory using the option `--reduce_paths` results in an actual path of

```
\foo\bar\baz\xyzy\xyzy\objects\file.c
```

3.156 --reinitialize_workdir

This option enables you to reinitialize the project template working directory set using `--workdir`.

When the directory set using `--workdir` refers to an existing working directory containing modified project template files, specifying this option causes the working directory to be deleted and recreated with new copies of the original project template files.

———— **Note** —————

This option is deprecated.

3.156.1 Restrictions

This option must be used in combination with the `--workdir` option.

3.156.2 See also

- `--project=filename`, `--no_project` on page 3-175
- `--workdir=directory` on page 3-227.

3.157 --relaxed_ref_def, --no_relaxed_ref_def

This option permits multiple object files to use tentative definitions of global variables. Some traditional programs are written using this declaration style.

3.157.1 Usage

This option is primarily provided for compatibility with GNU C. It is not recommended for new application code.

3.157.2 Default

The default is strict references and definitions. (Each global variable can only be declared in one object file.) However, if you specify an ARM Linux configuration file on the command line and you use `--translate_gcc`, the default is `--relaxed_ref_def`.

3.157.3 Restrictions

This option is not available in C++.

3.157.4 See also

- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--translate_gcc](#) on page 3-200
- *Rationale for International Standard - Programming Languages - C.*

3.158 --remarks

This option instructs the compiler to issue remark messages, such as warning of padding in structures.

3.158.1 Default

By default, the compiler does not issue remarks.

3.158.2 See also

- [--brief_diagnostics, --no_brief_diagnostics](#) on page 3-29
- [--diag_error=tag\[,tag,...\]](#) on page 3-70
- [--diag_remark=tag\[,tag,...\]](#) on page 3-71
- [--diag_style={arm|ide|gnu}](#) on page 3-72
- [--diag_suppress=tag\[,tag,...\]](#) on page 3-73
- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [--errors=filename](#) on page 3-86
- [-W](#) on page 3-220
- [--wrap_diagnostics, --no_wrap_diagnostics](#) on page 3-228.

3.159 --remove_unneeded_entities, --no_remove_unneeded_entities

These options control whether debug information is generated for all source symbols, or only for those source symbols actually used.

3.159.1 Usage

Use `--remove_unneeded_entities` to reduce the amount of debug information in an ELF object. Faster linkage times can also be achieved.

———— **Caution** ————

Although `--remove_unneeded_entities` can help to reduce the amount of debug information generated per file, it has the disadvantage of reducing the number of debug sections that are common to many files. This reduces the number of common debug sections that the linker is able to remove at final link time, and can result in a final debug image that is larger than necessary. For this reason, use `--remove_unneeded_entities` only when necessary.

3.159.2 Restrictions

The effects of these options are restricted to debug information.

3.159.3 Default

The default is `--no_remove_unneeded_entities`.

3.159.4 See also

- *The DWARF Debugging Standard*, <http://dwarfstd.org/>

3.160 --restrict, --no_restrict

This option enables or disables the use of the C99 keyword `restrict`.

———— **Note** —————

The alternative keywords `__restrict` and `__restrict__` are supported as synonyms for `restrict`. These alternative keywords are always available, regardless of the use of the `--restrict` option.

3.160.1 Default

When compiling ISO C99 source code, use of the C99 keyword `restrict` is enabled by default.

When compiling ISO C90 or ISO C++ source code, use of the C99 keyword `restrict` is disabled by default.

3.160.2 See also

- [restrict on page 4-15](#).

3.161 --retain=*option*

This option enables you to restrict the optimizations performed by the compiler.

3.161.1 Syntax

--retain=*option*

Where *option* is one of the following:

fns	prevents the removal of unused functions
inlinefns	prevents the removal of unused inline functions
noninlinefns	prevents the removal of unused non-inline functions
paths	prevents path-removing optimizations, such as <code>a b</code> transformed to <code>a b</code> . This supports <i>Modified Condition Decision Coverage</i> (MCDC) testing.
calls	prevents calls being removed, for example by inlining or tailcalling.
calls:distinct	prevents calls being merged, for example by cross-jumping (that is, common tail path merging).
libcalls	prevents calls to library functions being removed, for example by inline expansion.
data	prevents data being removed.
rodata	prevents read-only data being removed.
rwwdata	prevents read-write data being removed.
data:order	prevents data being reordered.

If *option* is unspecified, the compiler faults use of --retain.

3.161.2 Usage

This option might be useful when performing validation, debugging, and coverage testing. In most other cases, it is not required.

Using this option can have a negative effect on code size and performance.

3.161.3 See also

- [__attribute__\(\(nomerge\)\) function attribute on page 5-52](#)
- [__attribute__\(\(notailcall\)\) function attribute on page 5-55](#).

3.162 --rtti, --no_rtti

These options control support for the RTTI features `dynamic_cast` and `typeid` in C++.

3.162.1 Usage

Use `--no_rtti` to disable source-level RTTI features such as `dynamic_cast`.

———— **Note** —————

You are permitted to use `dynamic_cast` without `--rtti` in cases where RTTI is not required, such as dynamic cast to an unambiguous base, and dynamic cast to `(void *)`. If you try to use `dynamic_cast` without `--rtti` in cases where RTTI *is* required, the compiler generates an error.

3.162.2 Mode

These options are effective only if the source language is C++.

3.162.3 Default

The default is `--rtti`.

3.162.4 See also

- [--dllimport_runtime, --no_dllimport_runtime](#) on page 3-78
- [--rtti_data, --no_rtti_data](#) on page 3-186.

3.163 `--rtti_data`, `--no_rtti_data`

These options enable and disable the generation of C++ RTTI data.

3.163.1 Usage

Use `--no_rtti_data` to disable both source-level features and the generation of most RTTI data. Even if `--no_rtti_data` is set, RTTI data are generated for exceptions.

———— **Note** —————

In RVCT 4.0 and later, the GCC option `-fno-rtti` implies `--no_rtti_data` when using GCC command-line translation.

3.163.2 Mode

These options are effective only if the source language is C++.

3.163.3 Default

The default is `--rtti_data`.

3.163.4 See also

- [--dllimport_runtime](#), [--no_dllimport_runtime](#) on page 3-78
- [--exceptions](#), [--no_exceptions](#) on page 3-87
- [--rtti](#), [--no_rtti](#) on page 3-185
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202.

3.164 -S

This option instructs the compiler to output the disassembly of the machine code generated by the compiler to a file.

Unlike the `--asm` option, object modules are not generated. The name of the assembly output file defaults to `filename.s` in the current directory, where `filename` is the name of the source file stripped of any leading directory names. The default filename can be overridden with the `-o` option.

You can use `armasm` to assemble the output file and produce object code. The compiler adds ASSERT directives for command-line options such as AAPCS variants and byte order to ensure that compatible compiler and assembler options are used when re-assembling the output. You must specify the same AAPCS settings to both the assembler and the compiler.

3.164.1 See also

- [--apcs=qualifer..qualifier](#) on page 3-11
- [--asm](#) on page 3-24
- [-c](#) on page 3-31
- [--info=totals](#) on page 3-121
- [--interleave](#) on page 3-124
- [--list](#) on page 3-133
- [-o filename](#) on page 3-154
- *Assembler Guide*.

3.165 --shared

This option enables a shared library to be generated when building for ARM Linux with the `--arm_linux_paths` option. It enables the selection of libraries and initialization code suitable for use in a shared library, based on the ARM Linux configuration.

3.165.1 Restrictions

You must use this option in conjunction with `--arm_linux_paths` and `--apcs=/fpic`.

3.165.2 Example

Link two object files, `obj1.o` and `obj2.o`, into a shared library named `libexample.o`:

```
armcc --arm_linux_paths --arm_linux_config_file=my_config_file --shared -o libexample.so
obj1.o obj2.o
```

3.165.3 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--configure_gcc=path](#) on page 3-43
- [--configure_gcc_version=version](#) on page 3-44
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--gnu_defaults](#) on page 3-108
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries, --no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.166 --show_cmdline

This option shows how the compiler processes the command line. It can be useful when checking:

- the command line a build system is using
- how the compiler is interpreting the supplied command line, for example, the ordering of command-line options.

The commands are shown normalized, and the contents of any `via` files are expanded.

The output is sent to the standard output stream (`stdout`).

Note

If using this option with the ARM Linux translation options, you must use `-Warmcc`. For example, `armcc -Warmcc,--show_cmdline --translate_gcc ...`

3.166.1 See also

- [-Aopt](#) on page 3-6
- [--echo](#) on page 3-83
- [-Lopt](#) on page 3-127
- [--via=filename](#) on page 3-216.

3.167 --signed_bitfields, --unsigned_bitfields

This option makes bitfields of type `int` signed or unsigned.

The C Standard specifies that if the type specifier used in declaring a bitfield is either `int`, or a `typedef` name defined as `int`, then whether the bitfield is signed or unsigned is dependent on the implementation.

3.167.1 Default

The default is `--unsigned_bitfields`. However, if you specify an ARM Linux configuration file on the command line and you use `--translate_gcc` or `--translate_g++`, the default is `--signed_bitfields`.

Note

The AAPCS requirement for bitfields to default to unsigned on ARM, is relaxed in version 2.03 of the standard.

3.167.2 Example

```
typedef int integer;
struct
{
    integer x : 1;
} bf;
```

Compiling this code with `--signed_bitfields` causes to be treated as a signed bitfield.

3.167.3 See also

- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--gnu_defaults](#) on page 3-108
- *Procedure Call Standard for the ARM Architecture*,
<http://infocenter/help/index.jsp?topic=/com.arm.doc.ih0042->

3.168 --signed_chars, --unsigned_chars

This option makes the `char` type signed or unsigned.

When `char` is signed, the macro `__FEATURE_SIGNED_CHAR` is also defined by the compiler.

Note

- Care must be taken when mixing translation units that have been compiled with and without this option, and that share interfaces or data structures.
 - The ARM ABI defines `char` as an unsigned byte, and this is the interpretation used by the C++ libraries.
-

3.168.1 Default

The default is `--unsigned_chars`.

3.168.2 See also

- [Predefined macros on page 5-183](#).

3.169 --split_ldm

This option instructs the compiler to split LDM and STM instructions into two or more LDM or STM instructions.

When `--split_ldm` is selected, the maximum number of register transfers for an LDM or STM instruction is limited to:

- five, for all STMs
- five, for LDMs that do not load the PC
- four, for LDMs that load the PC.

Where register transfers beyond these limits are required, multiple LDM or STM instructions are used.

3.169.1 Usage

The `--split_ldm` option can be used to reduce interrupt latency on ARM systems that:

- do not have a cache or a write buffer, for example, a cacheless ARM7TDMI
- use zero-wait-state, 32-bit memory.

———— **Note** —————

Using `--split_ldm` increases code size and decreases performance slightly.

3.169.2 Restrictions

- Inline assembler LDM and STM instructions are split by default when `--split_ldm` is used. However, the compiler might subsequently recombine the separate instructions into an LDM or STM.
- Only LDM and STM instructions are split when `--split_ldm` is used.
- Some target hardware does not benefit from code built with `--split_ldm`. For example:
 - It has no significant benefit for cached systems, or for processors with a write buffer.
 - It has no benefit for systems with non zero-wait-state memory, or for systems with slow peripheral devices. Interrupt latency in such systems is determined by the number of cycles required for the slowest memory or peripheral access. Typically, this is much greater than the latency introduced by multiple register transfers.

3.169.3 See also

- [Inline assembler and instruction expansion in C and C++ code on page 8-21](#) in *Using the Compiler*.

3.170 --split_sections

This option instructs the compiler to generate one ELF section for each function in the source file.

Output sections are named with the same name as the function that generates the section, but with an `i.` prefix.

———— **Note** —————

If you want to place specific data items or structures in separate sections, mark them individually with `__attribute__((section(...)))`.

If you want to remove unused functions, it is recommended that you use the linker feedback optimization in preference to this option. This is because linker feedback produces smaller code by avoiding the overhead of splitting all sections.

3.170.1 Restrictions

This option reduces the potential for sharing addresses, data, and string literals between functions. Consequently, it might increase code size slightly for some functions.

3.170.2 Example

```
int f(int x)
{
    return x+1;
}
```

Compiling this code with `--split_sections` produces:

```
        AREA |i.f|, CODE, READONLY, ALIGN=2
f PROC
    ADD    r0,r0,#1
    BX    lr
    ENDP
```

3.170.3 See also

- [--data_reorder, --no_data_reorder](#) on page 3-55
- [--feedback=filename](#) on page 3-93
- [--multifile, --no_multifile](#) on page 3-150
- [__attribute__\(\(section\("name"\)\)\)](#) function attribute on page 5-58
- [#pragma arm section \[section_type_list\]](#) on page 5-88
- [Linker feedback during compilation on page 3-23](#) in *Using the Compiler*.

3.171 --strict, --no_strict

This option enforces or relaxes strict C or strict C++, depending on the choice of source language used.

When `--strict` is selected:

- features that conflict with ISO C or ISO C++ are disabled
- error messages are returned when nonstandard features are used.

3.171.1 Default

The default is `--no_strict`.

3.171.2 Usage

`--strict` enforces compliance with:

- | | |
|----------------|--|
| ISO C90 | • ISO/IEC 9899:1990, the 1990 International Standard for C. |
| | • ISO/IEC 9899 AM1, the 1995 Normative Addendum 1. |
| ISO C99 | ISO/IEC 9899:1999, the 1999 International Standard for C. |
| ISO C++ | ISO/IEC 14822:2003, the 2003 International Standard for C++. |

3.171.3 Errors

When `--strict` is in force and a violation of the relevant ISO standard occurs, the compiler issues an error message.

The severity of diagnostic messages can be controlled in the usual way.

3.171.4 Example

```
void foo(void)
{
    long long i; /* okay in nonstrict C90 */
}
```

Compiling this code with `--strict` generates an error.

3.171.5 See also

- [--c90](#) on page 3-33
- [--c99](#) on page 3-34
- [--cpp](#) on page 3-47
- [--gnu](#) on page 3-107
- [--strict_warnings](#) on page 3-195
- [Dollar signs in identifiers](#) on page 4-21
- [Source language modes of the compiler on page 2-3](#) in *Using the Compiler*.

3.172 --strict_warnings

Diagnostics that are errors in `--strict` mode are downgraded to warnings, where possible. It is sometimes not possible for the compiler to downgrade a strict error, for example, where it cannot construct a legitimate program to recover.

3.172.1 Errors

When `--strict_warnings` is in force and a violation of the relevant ISO standard occurs, the compiler normally issues a warning message.

The severity of diagnostic messages can be controlled in the usual way.

———— **Note** —————

In some cases, the compiler issues an error message instead of a warning when it detects something that is strictly illegal, and terminates the compilation. For example:

```
#ifdef $Super$
extern void $Super$$__aeabi_idiv0(void); /* intercept __aeabi_idiv0 */
#endif
```

Compiling this code with `--strict_warnings` generates an error if you do not use the `--dollar` option.

3.172.2 Example

```
void foo(void)
{
    long long i; /* okay in nonstrict C90 */
}
```

Compiling this code with `--strict_warnings` generates a warning message.

Compilation continues, even though the expression `long long` is strictly illegal.

3.172.3 See also

- [Source language modes of the compiler](#) on page 2-3
- [Dollar signs in identifiers](#) on page 4-21
- [--c90](#) on page 3-33
- [--c99](#) on page 3-34
- [--cpp](#) on page 3-47
- [--gnu](#) on page 3-107
- [--strict, --no_strict](#) on page 3-194.

3.173 --sys_include

This option removes the current place from the include search path.

Quoted include files are treated in a similar way to angle-bracketed include files, except that quoted include files are always searched for first in the directories specified by -I, and angle-bracketed include files are searched for first in the -J directories.

3.173.1 See also

- [-Idir\[,dir,...\]](#) on page 3-114
- [-Jdir\[,dir,...\]](#) on page 3-125
- [--kandr_include](#) on page 3-126
- [--preinclude=filename](#) on page 3-172
- [Compiler search rules and the current place on page 3-20](#) in *Using the ARM Compiler*
- [Compiler command-line options and search paths on page 3-19](#) in *Using the ARM Compiler*.

3.174 --thumb

This option configures the compiler to target the Thumb instruction set.

3.174.1 Default

This is the default option for targets that do not support the ARM instruction set.

3.174.2 See also

- [--arm on page 3-15](#)
- [#pragma arm on page 5-87](#)
- [#pragma thumb on page 5-112](#)
- [ARM architectures supported by the toolchain on page 2-17 in *Introducing the ARM Compiler toolchain*](#)
- [Selecting the target CPU at compile time on page 6-8 in *Using the Compiler*](#).

3.175 --translate_g++

This option helps to emulate the GNU compiler in C++ mode by enabling the translation of command lines from the GNU tools.

3.175.1 Usage

You can use this option to provide either of the following:

- a full GCC emulation targeting ARM Linux.
- a subset of full GCC emulation in the form of translating individual GCC command-line arguments into their ARM compiler equivalents.

To provide a full ARM Linux GCC emulation, you must also use `--arm_linux_config_file`. This combination of options selects the appropriate GNU header files and libraries specified by the configuration file, and includes changes to some default behaviors.

To translate GCC command-line arguments into their ARM compiler equivalents without aiming for full GCC emulation, use `--translate_g++` to emulate `g++`, but do not use it with `--arm_linux_config_file`. Because you are not aiming for full GCC emulation with this method, the default behavior of the ARM compilation tools is retained, and no defaults are set for targeting ARM Linux. The library paths and option defaults for the ARM compilation tools remained unchanged.

Specifying multiple GNU translation modes on the same command line is ambiguous to the compiler. The first specified translation mode is used, and the compiler generates a warning message. For example, given `armcc --translate_g++ --translate_gld`, the compiler uses `--translate_g++`, ignores `--translate_gld`, and generates a warning message.

If you specify an ARM Linux configuration file on the command line and you use `--translate_g++`, this alters the default settings for:

- `--exceptions`, `--no_exceptions`
- `--bss_threshold`
- `--relaxed_ref_def`, `--no_relaxed_ref_def`
- `--signed_bitfields`, `--unsigned_bitfields`.

To selectively specify options that are not to be translated, use `-warmcc`.

3.175.2 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--bss_threshold=num](#) on page 3-30
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--configure_gcc=path](#) on page 3-43
- [--configure_gcc_version=version](#) on page 3-44
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--exceptions](#), [--no_exceptions](#) on page 3-87
- [--gnu_defaults](#) on page 3-108

- [--relaxed_ref_def, --no_relaxed_ref_def](#) on page 3-180
- [--shared](#) on page 3-188
- [--signed_bitfields, --unsigned_bitfields](#) on page 3-190
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [-Warmcc,option\[,option,...\]](#) on page 3-221
- [-Warmcc,--gcc_fallback](#) on page 3-222
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries, --no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.176 --translate_gcc

This option helps to emulate gcc by enabling the translation of command lines from the GNU tools.

3.176.1 Usage

You can use this option to provide either of the following:

- a full GCC emulation targeting ARM Linux
- a subset of full GCC emulation in the form of translating individual GCC command-line arguments into their ARM compiler equivalents.

To provide a full GCC emulation, you must also use `--arm_linux_config_file`. This combination of options selects the appropriate GNU header files and libraries specified by the configuration file, and includes changes to some default behaviors.

To translate individual GCC command-line arguments into their ARM compiler equivalents without aiming for full GCC emulation, use `--translate_gcc` to emulate gcc, but do not use it with `--arm_linux_config_file`. Because you are not aiming for full GCC emulation with this method, the default behavior of the ARM compilation tools is retained, and no defaults are set for targeting ARM Linux. The library paths and option defaults for the ARM compilation tools remained unchanged.

To selectively specify options that are not to be translated, use `-warmcc`.

Specifying multiple GNU translation modes on the same command line is ambiguous to the compiler. The first specified translation mode is used, and the compiler generates a warning message. For example, given `armcc --translate_gcc --translate_gld`, the compiler uses `--translate_gcc`, ignores `--translate_gld`, and generates a warning message.

If you specify an ARM Linux configuration file on the command line and you use `--translate_gcc`, this alters the default settings for:

- `--bss_threshold`
- `--relaxed_ref_def`, `--no_relaxed_ref_def`
- `--signed_bitfields`, `--unsigned_bitfields`.

3.176.2 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--bss_threshold=num](#) on page 3-30
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--configure_gcc=path](#) on page 3-43
- [--configure_gcc_version=version](#) on page 3-44
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--gnu_defaults](#) on page 3-108
- [--relaxed_ref_def](#), [--no_relaxed_ref_def](#) on page 3-180
- [--shared](#) on page 3-188

- [--signed_bitfields, --unsigned_bitfields](#) on page 3-190
- [--translate_g++](#) on page 3-198
- [--translate_gld](#) on page 3-202
- [-Warmcc,option\[,option,...\]](#) on page 3-221
- [-Warmcc,--gcc_fallback](#) on page 3-222
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries, --no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.177 --translate_gld

This option helps to emulate GNU ld by enabling the translation of command lines from the GNU tools.

3.177.1 Usage

You can use this option to provide either of the following:

- a full GNU ld emulation targeting ARM Linux
- a subset of full GNU ld emulation in the form of translating individual GNU ld command-line arguments into their ARM compiler equivalents.

To provide a full GNU ld emulation, you must also use `--arm_linux_config_file`. This combination of options selects the appropriate GNU library paths specified by the configuration file, and includes changes to some default behaviors.

To translate individual GNU ld command-line arguments into their ARM compiler equivalents without aiming for full GNU ld emulation, use `--translate_gld` to emulate GNU ld, but do not use it with `--arm_linux_config_file`. Because you are not aiming for full GNU ld emulation with this method, default behavior of the ARM compilation tools is retained, and no defaults are set for targeting ARM Linux. The library paths and option defaults for the ARM compilation tools remained unchanged.

Note

- `--translate_gld` is used by invoking `armcc` as if it were the GNU linker. This is intended only for use by existing build scripts that involve the GNU linker directly.
 - In `gcc` and `g++` modes, `armcc` reports itself with `--translate_gld` as the linker it uses. For example, `gcc -print-file-name=ld`.
-

To selectively specify options that are not to be translated, use `-warmcc`.

Specifying multiple GNU translation modes on the same command line is ambiguous to the compiler. The first specified translation mode is used, and the compiler generates a warning message. For example, given `armcc --translate_gcc --translate_gld`, the compiler uses `--translate_gcc`, ignores `--translate_gld`, and generates a warning message.

3.177.2 See also

- [--arm_linux](#) on page 3-16
- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--arm_linux_paths](#) on page 3-21
- [--configure_cpp_headers=path](#) on page 3-39
- [--configure_extra_includes=paths](#) on page 3-40
- [--configure_extra_libraries=paths](#) on page 3-41
- [--configure_gcc=path](#) on page 3-43
- [--configure_gcc_version=version](#) on page 3-44
- [--configure_gld=path](#) on page 3-45
- [--configure_sysroot=path](#) on page 3-46
- [--gnu_defaults](#) on page 3-108
- [--shared](#) on page 3-188
- [--translate_g++](#) on page 3-198

- [--translate_gcc](#) on page 3-200
- [-Warmcc,option\[,option,...\]](#) on page 3-221
- [-Warmcc,--gcc_fallback](#) on page 3-222
- [--arm_linux](#) on page 2-13 in the *Linker Reference*
- [--library=name](#) on page 2-97 in the *Linker Reference*
- [--search_dynamic_libraries, --no_search_dynamic_libraries](#) on page 2-144 in the *Linker Reference*
- [Chapter 2 About building Linux applications with the ARM Compiler toolchain and GNU libraries](#) in *Building Linux Applications with the ARM Compiler toolchain and GNU Libraries*.

3.178 --trigraphs, --no_trigraphs

This option enables and disables trigraph recognition.

3.178.1 Default

The default is --trigraphs, except in GNU mode, where the default is --no_trigraphs.

3.178.2 See also

- *ISO/IEC 9899:TC2.*

3.179 --type_traits_helpers, --no_type_traits_helpers

These options enable and disable support for C++ type traits helpers (such as `__is_union` and `__has_virtual_destructor`). Type traits helpers are enabled in non-GNU C++ mode by default, and in GNU C++ mode when emulating g++ 4.3 and later.

3.179.1 See also

- [--gnu_version=version](#) on page 3-110.

3.180 -Uname

This option removes any initial definition of the macro *name*.

The macro *name* can be either:

- a predefined macro
- a macro specified using the -D option.

Note

Not all compiler predefined macros can be undefined.

3.180.1 Syntax

-Uname

Where:

name is the name of the macro to be undefined.

3.180.2 Usage

Specifying -Uname has the same effect as placing the text #undef *name* at the head of each source file.

3.180.3 Restrictions

The compiler defines and undefines macros in the following order:

1. compiler predefined macros
2. macros defined explicitly, using -Dname
3. macros explicitly undefined, using -Uname.

3.180.4 See also

- [-C on page 3-32](#)
- [-Dname\[\(parm-list\)\]\[=def\] on page 3-54](#)
- [-E on page 3-82](#)
- [-M on page 3-144](#)
- [Predefined macros on page 5-183.](#)

3.181 --unaligned_access, --no_unaligned_access

These options enable and disable unaligned accesses to data on ARM architecture-based processors.

3.181.1 Default

The default is `--unaligned_access` on ARM-architecture based processors that support unaligned accesses to data. This includes:

- all ARMv6 architecture-based processors
- ARMv7-A, ARMv7-R, and ARMv7-M architecture-based processors.

The default is `--no_unaligned_access` on ARM-architecture based processors that do not support unaligned accesses to data. This includes:

- all pre-ARMv6 architecture-based processors
- ARMv6-M architecture-based processors.

3.181.2 Usage

`--unaligned_access`

Use `--unaligned_access` on processors that support unaligned accesses to data, for example `--cpu=ARM1136J-S`, to speed up accesses to packed structures.

To enable unaligned support, you must:

- Clear the A bit, bit 1, of CP15 register 1 in your initialization code.
- Set the U bit, bit 22, of CP15 register 1 in your initialization code.

The initial value of the U bit is determined by the **UBITINIT** input to the core. The MMU must be on, and the memory marked as normal memory.

The libraries include special versions of certain library functions designed to exploit unaligned accesses. When unaligned access support is enabled, the compilation tools use these library functions to take advantage of unaligned accesses.

`--no_unaligned_access`

Use `--no_unaligned_access` to disable the generation of unaligned word and halfword accesses on ARMv6 processors.

To enable modulo four-byte alignment checking on an ARMv6 target without unaligned accesses, you must:

- Set the A bit, bit 1, of CP15 register 1 in your initialization code.
- Set the U bit, bit 22, of CP15 register 1 in your initialization code.

The initial value of the U bit is determined by the **UBITINIT** input to the core.

Note

Unaligned doubleword accesses, for example unaligned accesses to **long long** integers, are not supported by ARM processor cores. Doubleword accesses must be either eight-byte or four-byte aligned.

The compiler does not provide support for modulo eight-byte alignment checking. That is, the configuration $U = 0, A = 1$ in CP15 register 1 is not supported by the compiler, or more generally, by the ARM compiler toolset.

The libraries include special versions of certain library functions designed to exploit unaligned accesses. To prevent these enhanced library functions being used when unaligned access support is disabled, you have to specify `--no_unaligned_access` on both the compiler command line and the assembler command line when compiling a mixture of C and C++ source files and assembly language source files.

3.181.3 Restrictions

Code compiled for processors supporting unaligned accesses to data can run correctly only if the choice of alignment support in software matches the choice of alignment support on the processor core.

3.181.4 See also

- [--cpu=name](#) on page 3-49
- [Assembler command line syntax on page 2-2](#) in the *Assembler Reference*
- [ARM architecture v4T on page 2-11](#) in *Developing Software for ARM Processors*.

3.182 --use_frame_pointer

This option causes armcc to set the frame pointer to the current stack frame. Using this option reserves R11 to store the frame pointer in ARM and Thumb code.

3.182.1 See also

- [ARM registers on page 3-9](#) in Using the Assembler
- [General-purpose registers on page 3-11](#) in Using the Assembler.

3.183 --use_gas

This option invokes the *GNU assembler* (`gas`) rather than `armasm` when you compile source files ending in `.s` or `.S` filename extensions. It is only applicable when using GNU translation (`-Warmcc`).

3.183.1 Usage

During translation, invoke `gas` with `-Warmcc, --use_gas`.

3.183.2 See also

- [--configure_gas=path](#) on page 3-42
- [-Warmcc,option\[,option,...\]](#) on page 3-221.

3.184 --use_pch=*filename*

This option instructs the compiler to use a PCH file with the specified filename as part of the current compilation.

This option takes precedence if you include --pch on the same command line.

3.184.1 Syntax

--use_pch=*filename*

Where:

filename is the PCH file to be used as part of the current compilation.

3.184.2 Restrictions

The effect of this option is negated if you include --create_pch=*filename* on the same command line.

3.184.3 Errors

If the specified file does not exist, or is not a valid PCH file, the compiler generates an error.

3.184.4 See also

- [--create_pch=filename](#) on page 3-53
- [--pch](#) on page 3-165
- [--pch_dir=dir](#) on page 3-166
- [--pch_messages, --no_pch_messages](#) on page 3-167
- [--pch_verbose, --no_pch_verbose](#) on page 3-168
- [PreCompiled Header \(PCH\) files](#) on page 5-33 in *Using the Compiler*.

3.185 --using_std, --no_using_std

This option enables or disables implicit use of the std namespace when standard header files are included in C++.

———— **Note** —————

This option is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

3.185.1 Mode

This option is effective only if the source language is C++.

3.185.2 Default

The default is --no_using_std.

3.185.3 See also

- [Namespaces on page 6-20.](#)

3.186 --vectorize, --no_vectorize

This option enables or disables the generation of NEON vector instructions directly from C or C++ code.

3.186.1 Default

The default is `--no_vectorize`.

3.186.2 Restrictions

The following options must be specified for loops to vectorize:

- `--cpu=name` Target processor must have NEON capability.
- `-Otime` Type of optimization to reduce execution time.
- `-Onum` Level of optimization. One of the following must be used:
 - `-O2` High optimization. This is the default.
 - `-O3` Maximum optimization.

Note

NEON is an implementation of the ARM Advanced *Single Instruction, Multiple Data* (SIMD) extension.

A separate *FLEXnet* license is needed to enable the use of vectorization.

3.186.3 Example

This example invokes automatic vectorization on the Cortex-A8 processor.

```
armcc --vectorize --cpu=Cortex-A8 -O3 -Otime -c file.c
```

Using the command-line options `-O3` and `-Otime` ensures that the code achieves significant performance benefits in addition to those of vectorization.

Note

You can also compile with `-O2 -Otime`. However, this does not give the maximum code performance.

3.186.4 See also

- [--cpu=name](#) on page 3-49
- [-Onum](#) on page 3-156
- [-Otime](#) on page 3-161
- [--reassociate_saturation, --no_reassociate_saturation](#) on page 3-177
- [Introducing NEON Development Article](#), <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002->
- [Chapter 4 Using the NEON Vectorizing Compiler](#) in *Using the ARM Compiler*
- [Licensed features of the toolchain](#) on page 2-10 in *Introducing the ARM Compiler toolchain*.

3.187 --version_number

This option displays the version of armcc being used.

3.187.1 Syntax

```
armcc --version_number
```

The compiler displays the version number in the format nnnbbb, where:

- nnn is the version number
- bbbb is the build number.

3.187.2 Example

Version 5.01 build 0019 is displayed as 5010019.

3.187.3 See also

- [--help](#) on page 3-112
- [--vsn](#) on page 3-219.

3.188 --vfe, --no_vfe

This option enables or disables *Virtual Function Elimination* (VFE) in C++.

VFE enables unused virtual functions to be removed from code. When VFE is enabled, the compiler places the information in special sections with the prefix `.arm_vfe_`. These sections are ignored by linkers that are not VFE-aware, because they are not referenced by the rest of the code. Therefore, they do not increase the size of the executable. However, they increase the size of the object files.

3.188.1 Mode

This option is effective only if the source language is C++.

3.188.2 Default

The default is `--vfe`, except for the case where legacy object files compiled with a pre-RVCT v2.1 compiler do not contain VFE information.

3.188.3 See also

- [Calling a pure virtual function on page E-3](#)
- [Elimination of unused virtual functions on page 5-6](#) in *Using the Linker*.

3.189 --via=*filename*

This option instructs the compiler to read additional command-line options from a specified file. The options read from the file are added to the current command line.

Via commands can be nested within via files.

3.189.1 Syntax

```
--via=filename
```

Where:

filename is the name of a via file containing options to be included on the command line.

If *filename* is unspecified, the compiler faults use of --via.

3.189.2 Example

Given a source file `main.c`, a via file `apcs.txt` containing the line:

```
--apcs=/rwp_i --no_lower_rwp_i --via=L_apcs.txt
```

and a second via file `L_apcs.txt` containing the line:

```
-L--rwp_i -L--callgraph
```

compiling `main.c` with the command line:

```
armcc main.c -L-o"main.axf" --via=apcs.txt
```

compiles `main.c` using the command line:

```
armcc --no_lower_rwp_i --apcs=/rwp_i -L--rwp_i -L--callgraph -L-o"main.axf" main.c
```

3.189.3 See also

- [Appendix B Via File Syntax](#)
- [Using a text file to specify command-line options](#) on page 2-24 in *Introducing ARM Compilation Tools*.

3.190 `--visibility_inlines_hidden`

This option stops inline member functions acquiring dynamic linkage (default visibility) from:

- `class __declspec(dllexport)`
- a class visibility attribute
- `--no_hide_all`.

Non-member functions are not affected.

3.190.1 See also

- [__declspec\(dllexport\) on page 5-31](#)
- [__attribute__\(\(visibility\("visibility_type"\)\)\) function attribute on page 5-61](#)
- [--hide_all, --no_hide_all on page 3-113](#).

3.191 --vla, --no_vla

This option enables or disables support for variable length arrays.

3.191.1 Default

C90 and Standard C++ do not support variable length arrays by default. Select the option `--vla` to enable support for variable length arrays in C90 or Standard C++.

Variable length arrays are supported both in Standard C and the GNU compiler extensions. The option `--vla` is implicitly selected either when the source language is C99 or the option `--gnu` is specified.

3.191.2 Example

```
size_t arr_size(int n)
{
    char array[n];           // variable length array, dynamically allocated
    return sizeof array;    // evaluated at runtime
}
```

3.191.3 See also

- [--c90](#) on page 3-33
- [--c99](#) on page 3-34
- [--cpp](#) on page 3-47
- [--gnu](#) on page 3-107.

3.192 --vsn

This option displays the version information and the license details. For example:

```
>armcc --vsn
ARM C/C++ Compiler, N.nn [Build num]
license_type
Software supplied by: ARM Limited
```

3.192.1 See also

- [--help](#) on page 3-112
- [--version_number](#) on page 3-214.

3.193 -w

This option instructs the compiler to suppress all warning messages.

3.193.1 See also

- [--brief_diagnostics, --no_brief_diagnostics](#) on page 3-29
- [--diag_error=tag\[,tag,...\]](#) on page 3-70
- [--diag_remark=tag\[,tag,...\]](#) on page 3-71
- [--diag_style={arm|ide|gnu}](#) on page 3-72
- [--diag_suppress=tag\[,tag,...\]](#) on page 3-73
- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [--errors=filename](#) on page 3-86
- [--remarks](#) on page 3-181
- [--wrap_diagnostics, --no_wrap_diagnostics](#) on page 3-228

3.194 `-Warmcc,option[,option,...]`

This option enables normal compiler command-line options to be passed to the compiler in GCC emulation mode. The options associated with `-Warmcc` are passed verbatim to the compiler, that is, without translation. These options also override any translation options that are specified.

3.194.1 Syntax

```
-Warmcc,option[,option,...]
```

Where:

`option` is a normal ARM compiler option.

3.194.2 Usage

Use this option to take advantage of features specific to the ARM compilation tools when in GCC emulation mode.

3.194.3 Example

In this example, `-Warmcc` enables the command-line options `-A` and `-L` to be used for passing options to the assembler and the linker without translation, while in GCC emulation mode.

```
armcc --translate_gcc --arm_linux_config_file=linux_config -o example.axf example.s
-Warmcc,-A--predefine="my_variable SETA 20" -Warmcc,-L--inline
```

3.194.4 See also

- [--arm_linux_config_file=path](#) on page 3-18
- [-Aopt](#) on page 3-6
- [-Lopt](#) on page 3-127
- [--translate_g++](#) on page 3-198
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [--inline, --no_inline](#) on page 2-85 in the *Linker Reference*
- [--predefine "directive"](#) on page 2-69 in the *Assembler Reference*.

3.195 `-warmcc,--gcc_fallback`

This option instructs the compiler to use GCC to retry a failed build step, when building for ARM Linux.

3.195.1 Usage

When using `armcc` in GCC emulation mode, GCC incompatibilities might cause a compile, assembly or link step to fail. Using this option instructs the compiler to automatically retry the failed step using GCC. Any build step that succeeds with the `armcc` does not get rebuilt using GCC. Each failed step is retried using GCC. For example, if you specify this option for all of the source files in your build and one of them contains an unsupported GNU extension, such as inline assembly code with the GCC syntax, `armcc` generates a warning and the compiler retries the failed command lines using the GNU tools.

———— **Note** —————

You must escape the option using `-warmcc`, for example `-warmcc,--gcc_fallback`.

3.195.2 Restrictions

This option can only be used with a GNU emulation mode (that is when using `--translate_gcc`, `--translate_g++`, or `--translate_gld`) and an ARM Linux configuration file specified with `--arm_linux_config_file`. An existing GNU toolchain must be present (either automatically found on the `PATH` environment variable or specified with `--configure_gcc`) to create the configuration file.

3.195.3 Example

```
armcc -c --translate_gcc --arm_linux_config_file=linux_config -warmcc,--gcc_fallback -o
example.o example.c
```

3.195.4 See also

- [--translate_g++ on page 3-198](#)
- [--translate_gcc on page 3-200](#)
- [--translate_gld on page 3-202](#)
- [--arm_linux_config_file=path on page 3-18](#)
- [--arm_linux_paths on page 3-21](#)
- [--configure_gcc=path on page 3-43](#)
- [--arm_linux_configure on page 3-19.](#)

Using the Compiler:

- [Using GCC fallback when building applications on page 3-24.](#)

3.196 --wchar, --no_wchar

This option permits or forbids the use of `wchar_t`. It does not necessarily fault declarations, providing they are unused.

3.196.1 Usage

Use this option to create an object file that is independent of `wchar_t` size.

3.196.2 Restrictions

If `--no_wchar` is specified:

- `wchar_t` fields in structure declarations are faulted by the compiler, regardless of whether or not the structure is used
- `wchar_t` in a typedef is faulted by the compiler, regardless of whether or not the typedef is used.

3.196.3 Default

The default is `--wchar`.

3.196.4 See also

- [--wchar16](#) on page 3-224
- [--wchar32](#) on page 3-225.

3.197 --wchar16

This option changes the type of `wchar_t` to **unsigned short**.

Selecting this option modifies both the type of the defined type `wchar_t` in C and the type of the native type `wchar_t` in C++. It also affects the values of `WCHAR_MIN` and `WCHAR_MAX`.

3.197.1 Default

The compiler assumes `--wchar16` unless `--wchar32` is explicitly specified.

3.197.2 See also

- [--wchar, --no_wchar](#) on page 3-223
- [--wchar32](#) on page 3-225
- [Predefined macros](#) on page 5-183.

3.198 --wchar32

This option changes the type of `wchar_t` to **unsigned int**.

Selecting this option modifies both the type of the defined type `wchar_t` in C and the type of the native type `wchar_t` in C++. It also affects the values of `WCHAR_MIN` and `WCHAR_MAX`.

3.198.1 Default

The compiler assumes `--wchar16` unless `--wchar32` is explicitly specified, or unless you specify an ARM Linux configuration file on the command line. Specifying an ARM Linux configuration file on the command line turns `--wchar32` on.

3.198.2 See also

- [--arm_linux_config_file=path](#) on page 3-18
- [--arm_linux_configure](#) on page 3-19
- [--gnu_defaults](#) on page 3-108
- [--wchar, --no_wchar](#) on page 3-223
- [--wchar16](#) on page 3-224
- [Predefined macros](#) on page 5-183.

3.199 --whole_program

This option promises the compiler that the source files specified on the command line form the whole program. The compiler is then able to apply optimizations based on the knowledge that the source code visible to it is the complete set of source code for the program being compiled. Without this knowledge, the compiler is more conservative when applying optimizations to the code.

3.199.1 Usage

Use this option to gain maximum performance from a small program.

3.199.2 Restriction

Do not use this option if you do not have all of the source code to give to the compiler.

3.199.3 See also

- [--multifile, --no_multifile](#) on page 3-150.

3.200 `--workdir=directory`

This option enables you to provide a working directory for a project template.

Note

Project templates only require working directories if they include files, for example, RVD configuration files.

Note

This option is deprecated.

3.200.1 Syntax

`--workdir=directory`

Where:

directory is the name of the project directory.

3.200.2 Restrictions

If you specify a project working directory using `--workdir`, then you must specify a project file using `--project`.

3.200.3 Errors

An error message is produced if you try to use `--project` without `--workdir` and `--workdir` is required.

3.200.4 See also

- [--project=filename, --no_project](#) on page 3-175
- [--reinitialize_workdir](#) on page 3-179.

3.201 --wrap_diagnostics, --no_wrap_diagnostics

This option enables or disables the wrapping of error message text when it is too long to fit on a single line.

3.201.1 Default

The default is `--no_wrap_diagnostics`.

3.201.2 See also

- [--brief_diagnostics, --no_brief_diagnostics](#) on page 3-29
- [--diag_error=tag\[,tag,...\]](#) on page 3-70
- [--diag_remark=tag\[,tag,...\]](#) on page 3-71
- [--diag_style={arm|ide|gnu}](#) on page 3-72
- [--diag_suppress=tag\[,tag,...\]](#) on page 3-73
- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [--errors=filename](#) on page 3-86
- [--remarks](#) on page 3-181
- [-W](#) on page 3-220
- [Chapter 7 Compiler Diagnostic Messages](#) in *Using the Compiler*.

Chapter 4

Language Extensions

This chapter describes the language extensions supported by the compiler:

- *Preprocessor extensions* on page 4-3
 - *#assert* on page 4-4
 - *#include_next* on page 4-5
 - *#unassert* on page 4-6
 - *#warning* on page 4-7
- *C99 language features available in C90* on page 4-8
 - *// comments* on page 4-9
 - *Subscripting struct* on page 4-10
 - *Flexible array members* on page 4-11
- *C99 language features available in C++ and C90* on page 4-12
 - *Variadic macros* on page 4-13
 - *long long* on page 4-14
 - *restrict* on page 4-15
 - *Hexadecimal floats* on page 4-16
- *Standard C language extensions* on page 4-17
 - *Constant expressions* on page 4-18
 - *Array and pointer extensions* on page 4-19
 - *Block scope function declarations* on page 4-20
 - *Dollar signs in identifiers* on page 4-21
 - *Top-level declarations* on page 4-22
 - *Benign redeclarations* on page 4-23
 - *External entities* on page 4-24

- *Function prototypes* on page 4-25
- *Standard C++ language extensions* on page 4-26
 - *? operator* on page 4-27
 - *Declaration of a class member* on page 4-28
 - *friend* on page 4-29
 - *Read/write constants* on page 4-30
 - *Scalar type constants* on page 4-32
 - *Specialization of nonmember function templates* on page 4-33
 - *Type conversions* on page 4-34
- *Standard C and Standard C++ language extensions* on page 4-35
 - *Address of a register variable* on page 4-36
 - *Arguments to functions* on page 4-37
 - *Anonymous classes, structures and unions* on page 4-38
 - *Assembler labels* on page 4-39
 - *Empty declaration* on page 4-40
 - *Hexadecimal floating-point constants* on page 4-41
 - *Incomplete enums* on page 4-42
 - *Integral type extensions* on page 4-43
 - *Label definitions* on page 4-44
 - *Long float* on page 4-45
 - *Nonstatic local variables* on page 4-46
 - *Structure, union, enum, and bitfield extensions* on page 4-47
- *GNU extensions to the C and C++ languages* on page 4-48.

For additional reference material on the compiler see also:

- *Appendix D Standard C Implementation Definition*
- *Appendix E Standard C++ Implementation Definition*
- *Appendix F C and C++ Compiler Implementation Limits.*

4.1 Preprocessor extensions

The compiler supports several extensions to the preprocessor, including the `#assert` preprocessing extensions of System V release 4.

See:

- [#assert](#) on page 4-4
- [#include_next](#) on page 4-5
- [#unassert](#) on page 4-6
- [#warning](#) on page 4-7.

4.2 #assert

The #assert preprocessing extensions of System V release 4 are permitted. These enable definition and testing of predicate names.

Such names are in a namespace distinct from all other names, including macro names.

4.2.1 Syntax

```
#assert name
```

```
#assert name[(token-sequence)]
```

Where:

name is a predicate name

token-sequence is an optional sequence of tokens.

If the token sequence is omitted, *name* is not given a value.

If the token sequence is included, *name* is given the value *token-sequence*.

4.2.2 Example

A predicate name defined using #assert can be tested in a #if expression, for example:

```
#if #name(token-sequence)
```

This has the value 1 if a #assert of the name *name* with the token-sequence *token-sequence* has appeared, and 0 otherwise. A given predicate can be given more than one value at a given time.

4.2.3 See also

- [#unassert on page 4-6.](#)

4.3 #include_next

This preprocessor directive is a variant of the #include directive. It searches for the named file only in the directories on the search path that follow the directory where the current source file is found, that is, the one containing the #include_next directive.

———— **Note** —————

This preprocessor directive is a GNU compiler extension that the ARM compiler supports.

4.4 #unassert

A predicate name can be deleted using a #unassert preprocessing directive.

4.4.1 Syntax

```
#unassert name
```

```
#unassert name[(token-sequence)]
```

Where:

name is a predicate name

token-sequence is an optional sequence of tokens.

If the token sequence is omitted, all definitions of *name* are removed.

If the token sequence is included, only the indicated definition is removed.

All other definitions are left intact.

4.4.2 See also

- [#assert](#) on page 4-4.

4.5 #warning

The preprocessing directive `#warning` is supported. Like the `#error` directive, this produces a user-defined warning at compilation time. However, it does not halt compilation.

4.5.1 Restrictions

The `#warning` directive is not available if the `--strict` option is specified. If used, it produces an error.

4.5.2 See also

- [--strict, --no_strict](#) on page 3-194.

4.6 C99 language features available in C90

The compiler supports numerous extensions to the ISO C90 standard, for example, C99-style `//` comments.

These extensions are available if the source language is C90 and you are compiling in nonstrict mode.

These extensions are not available if the source language is C90 and the compiler is restricted to compiling strict C90 using the `--strict` compiler option.

———— **Note** —————

Language features of Standard C and Standard C++, for example C++-style `//` comments, might be similar to the C90 language extensions. Such features continue to remain available if you are compiling strict Standard C or strict Standard C++ using the `--strict` compiler option.

See:

- [// comments](#) on page 4-9
- [Subscripting struct](#) on page 4-10
- [Flexible array members](#) on page 4-11.

4.7 // comments

The character sequence `//` starts a one line comment, like in C99 or C++.

`//` comments in C90 have the same semantics as `//` comments in C99.

4.7.1 Example

```
// this is a comment
```

4.7.2 See also

- [New language features of C99 on page 6-85](#) in *Using the Compiler*.

4.8 Subscripting struct

In C90, arrays that are not lvalues still decay to pointers, and can be subscripted. However, you must not modify or use them after the next sequence point, and you must not apply the unary & operator to them. Arrays of this kind can be subscripted in C90, but they do not decay to pointers outside C99 mode.

4.8.1 Example

```
struct Subscripting_Struct
{
    int a[4];
};
extern struct Subscripting_Struct Subscripting_0(void);
int Subscripting_1 (int index)
{
    return Subscripting_0().a[index];
}
```

4.9 Flexible array members

The last member of a **struct** can have an incomplete array type. The last member must not be the only member of the **struct**, otherwise the **struct** is zero in size.

4.9.1 Example

```
typedef struct
{
    int len;
    char p[]; // incomplete array type, for use in a malloc'd data structure
} str;
```

4.9.2 See also

- [New language features of C99 on page 6-85](#) in *Using the Compiler*.

4.10 C99 language features available in C++ and C90

The compiler supports numerous extensions to the ISO C++ standard and to the C90 language, for example, function prototypes that override old-style nonprototype definitions.

These extensions are available if:

- the source language is C++ and you are compiling in nonstrict mode
- the source language is C90 and you are compiling in nonstrict mode.

These extensions are not available if:

- the source language is C++ and the compiler is restricted to compiling strict Standard C++ using the `--strict` compiler option.
- the source language is C90 and the compiler is restricted to compiling strict Standard C using the `--strict` compiler option.

Note

Language features of Standard C, for example **long long** integers, might be similar to the C++ and C90 language extensions. Such features continue to remain available if you are compiling strict Standard C++ or strict C90 using the `--strict` compiler option.

See:

- [Variadic macros](#) on page 4-13
- [long long](#) on page 4-14
- [restrict](#) on page 4-15
- [Hexadecimal floats](#) on page 4-16.

4.11 Variadic macros

In C90 and C++ you can declare a macro to accept a variable number of arguments.

The syntax for declaring a variadic macro in C90 and C++ follows the C99 syntax for declaring a variadic macro, unless the option `--gnu` is selected. If the option `--gnu` is specified, the syntax follows GNU syntax for variadic macros.

4.11.1 Example

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
void variadic_macros(void)
{
    debug ("a test string is printed out along with %x %x %x\n", 12, 14, 20);
}
```

4.11.2 See also

- [--gnu on page 3-107](#)
- [New language features of C99 on page 6-85](#) in *Using the Compiler*.

4.12 long long

The ARM compiler supports 64-bit integer types through the type specifiers **long long** and **unsigned long long**. They behave analogously to **long** and **unsigned long** with respect to the usual arithmetic conversions. `__int64` is a synonym for **long long**.

Integer constants can have:

- an `ll` suffix to force the type of the constant to **long long**, if it fits, or to **unsigned long long** if it does not fit
- a `ull` or `llu` suffix to force the type of the constant to **unsigned long long**.

Format specifiers for `printf()` and `scanf()` can include `ll` to specify that the following conversion applies to a **long long** argument, as in `%lld` or `%llu`.

Also, a plain integer constant is of type **long long** or **unsigned long long** if its value is large enough. There is a warning message from the compiler indicating the change. For example, in strict 1990 ISO Standard C 2147483648 has type **unsigned long**. In ARM C and C++ it has the type **long long**. One consequence of this is the value of an expression such as:

```
2147483648 > -1
```

This expression evaluates to 0 in strict C and C++, and to 1 in ARM C and C++.

The **long long** types are accommodated in the usual arithmetic conversions.

4.12.1 See also

- [__int64](#) on page 5-14.

4.13 restrict

The **restrict** keyword is a C99 feature. It enables you to convey a declaration of intent to the compiler that different pointers and function parameter arrays do not point to overlapping regions of memory at runtime. This enables the compiler to perform optimizations that can otherwise be prevented because of possible aliasing.

4.13.1 Usage

The keywords `__restrict` and `__restrict__` are supported as synonyms for **restrict** and are always available.

You can specify `--restrict` to allow the use of the **restrict** keyword in C90 or C++.

4.13.2 Restrictions

The declaration of intent is effectively a promise to the compiler that, if broken, results in undefined behavior.

4.13.3 Example

The following example shows use of the **restrict** keyword applied to function parameter arrays.

```
void copy_array(int n, int *restrict a, int *restrict b)
{
    while (n-- > 0)
        *a++ = *b++;
}
```

The following example shows use of the **restrict** keyword applied to different pointers that exist in the form of local variables.

```
void copy_bytes(int n, int *a, int *b)
{
    int *restrict x;
    int *restrict y;

    x = a;
    y = b;

    while (n-- > 0)
        *x++ = *y++;
}
```

4.13.4 See also

- [--restrict, --no_restrict](#) on page 3-183
- [New language features of C99](#) on page 6-85 in *Using the Compiler*.

4.14 Hexadecimal floats

C90 and C++ support floating-point numbers that can be written in hexadecimal format.

4.14.1 Example

```
float hex_floats(void)
{
    return 0x1.fp3;    // 1.55e1
}
```

4.14.2 See also

- [New language features of C99 on page 6-85](#) in *Using the Compiler*.

4.15 Standard C language extensions

The compiler supports numerous extensions to the ISO C99 standard, for example, function prototypes that override old-style nonprototype definitions.

These extensions are available if:

- the source language is C99 and you are compiling in nonstrict mode
- the source language is C90 and you are compiling in nonstrict mode.

None of these extensions is available if:

- the source language is C90 and the compiler is restricted to compiling strict C90 using the `--strict` compiler option.
- the source language is C99 and the compiler is restricted to compiling strict Standard C using the `--strict` compiler option.
- the source language is C++.

See:

- [Constant expressions](#) on page 4-18
- [Array and pointer extensions](#) on page 4-19
- [Block scope function declarations](#) on page 4-20
- [Dollar signs in identifiers](#) on page 4-21
- [Top-level declarations](#) on page 4-22
- [Benign redeclarations](#) on page 4-23
- [External entities](#) on page 4-24
- [Function prototypes](#) on page 4-25.

4.16 Constant expressions

Extended constant expressions are supported in initializers. The following examples show the compiler behavior for the default, `--strict_warnings`, and `--strict` compiler modes.

4.16.1 Example 1, assigning the address of variable

Your code might contain constant expressions that assign the address of a variable at file scope, for example:

```
int i;
int j = (int)&i; /* but not allowed by ISO */
```

When compiling for C, this produces the following behavior:

- In default mode a warning is produced.
- In `--strict_warnings` mode a warning is produced.
- In `--strict` mode, an error is produced.

4.16.2 Example 2, constant value initializers

[Table 4-1](#) compares the behavior of the ARM compilation tools with the ISO C Standard.

If compiling with `--strict_warnings` in place of `--strict`, the example source code that is not valid with `--strict` become valid. The `--strict` error message is downgraded to a warning message.

Table 4-1 Behavior of constant value initializers in comparison with ISO Standard C

Example source code	ISO C Standard	ARM compilation tools	
		<code>--strict mode</code>	Nonstrict mode
<code>extern int const c = 10;</code>	Valid	Valid	Valid
<code>extern int const x = c + 10;</code>	Not valid	Not valid	Valid
<code>static int y = c + 10;</code>	Not valid	Not valid	Valid
<code>static int const z = c + 10;</code>	Not valid	Not valid	Valid
<code>extern int *const cp = (int*)0x100;</code>	Valid	Valid	Valid
<code>extern int *const xp = cp + 0x100;</code>	Not valid	Not valid	Valid
<code>static int *yp = cp + 0x100;</code>	Not valid	Not valid	Valid
<code>static int *const zp = cp + 0x100;</code>	Not valid	Not valid	Valid

4.16.3 See also

- [--extended_initializers, --no_extended_initializers](#) on page 3-92
- [--strict, --no_strict](#) on page 3-194
- [--strict_warnings](#) on page 3-195.

4.17 Array and pointer extensions

The following array and pointer extensions are supported:

- Assignment and pointer differences are permitted between pointers to types that are interchangeable but not identical, for example, **unsigned char *** and **char ***. This includes pointers to same-sized integral types, typically, **int *** and **long ***. A warning is issued. Assignment of a string constant to a pointer to any kind of character is permitted without a warning.
- Assignment of pointer types is permitted in cases where the destination type has added type qualifiers that are not at the top level, for example, assigning **int **** to **const int ****. Comparisons and pointer difference of such pairs of pointer types are also permitted. A warning is issued.
- In operations on pointers, a pointer to **void** is always implicitly converted to another type if necessary. Also, a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO C, some operators permit these, and others do not.
- Pointers to different function types can be assigned or compared for equality (**==**) or inequality (**!=**) without an explicit type cast. A warning or error is issued. This extension is prohibited in C++ mode.
- A pointer to **void** can be implicitly converted to, or from, a pointer to a function type.
- In an initializer, a pointer constant value can be cast to an integral type if the integral type is big enough to contain it.
- A non lvalue array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.

4.18 Block scope function declarations

Two extensions to block scope function declarations are supported:

- a block-scope function declaration also declares the function name at file scope
- a block-scope function declaration can have static storage class, thereby causing the resulting declaration to have internal linkage by default.

4.18.1 Example

```
void f1(void)
{
    static void g(void); /* static function declared in local scope */
                          /* use of static keyword is illegal in strict ISO C */
}
void f2(void)
{
    g();                 /* uses previous local declaration */
}
static void g(int i)
{ } /* error - conflicts with previous declaration of g */
```

4.19 Dollar signs in identifiers

Dollar (\$) signs are permitted in identifiers.

———— **Note** —————

When compiling with the `--strict` option, you can use the `--dollar` command-line option to permit dollar signs in identifiers.

4.19.1 Example

```
#define DOLLAR$
```

4.19.2 See also

- [--dollar, --no_dollar](#) on page 3-79
- [--strict, --no_strict](#) on page 3-194.

4.20 Top-level declarations

A C input file can contain no top-level declarations.

4.20.1 Errors

A remark is issued if a C input file contains no top-level declarations.

———— **Note** —————

Remarks are not displayed by default. To see remark messages, use the compiler option `--remarks`.

4.20.2 See also

- [--remarks on page 3-181](#).

4.21 Benign redeclarations

Benign redeclarations of **typedef** names are permitted. That is, a **typedef** name can be redeclared in the same scope as the same type.

4.21.1 Example

```
typedef int INT;typedef int INT; /* redeclaration */
```

4.22 External entities

External entities declared in other scopes are visible.

4.22.1 Errors

The compiler generates a warning if an external entity declared in another scope is visible.

4.22.2 Example

```
void f1(void)
{
    extern void f();
}
void f2(void)
{
    f(); /* Out of scope declaration */
}
```

4.23 Function prototypes

The compiler recognizes function prototypes that override old-style nonprototype definitions that appear at a later position in your code.

4.23.1 Errors

The compiler generates a warning message if you use old-style function prototypes.

4.23.2 Example

```
int function_prototypes(char);  
// Old-style function definition.  
int function_prototypes(x)  
    char x;  
{  
    return x == 0;  
}
```

4.24 Standard C++ language extensions

The compiler supports numerous extensions to the ISO C++ standard, for example, qualified names in the declaration of class members.

These extensions are available if the source language is C++ and you are compiling in nonstrict mode.

These extensions are not available if the source language is C++ and the compiler is restricted to compiling strict Standard C++ using the `--strict` compiler option.

See:

- [? operator on page 4-27](#)
- [Declaration of a class member on page 4-28](#)
- [friend on page 4-29](#)
- [Read/write constants on page 4-30](#)
- [Scalar type constants on page 4-32](#)
- [Specialization of nonmember function templates on page 4-33](#)
- [Type conversions on page 4-34.](#)

4.25 ? operator

A ? operator whose second and third operands are string literals or wide string literals can be implicitly converted to `char *` or `wchar_t *`. In C++ string literals are `const`. There is an implicit conversion that enables conversion of a string literal to `char *` or `wchar_t *`, dropping the `const`. That conversion, however, applies only to simple string literals. Permitting it for the result of a ? operation is an extension.

4.25.1 Example

```
char *p = x ? "abc" : "def";
```


4.26 Declaration of a class member

A qualified name can be used in the declaration of a class member.

4.26.1 Errors

A warning is issued if a qualified name is used in the declaration of a class member.

4.26.2 Example

```
struct A
{
    int A::f(); // is the same as int f();
};
```

4.27 friend

A **friend** declaration for a **class** can omit the class keyword.

Access checks are not carried out on **friend** declarations by default. Use the `--strict` command-line option to force access checking.

4.27.1 Example

```
class B;
class A
{
    friend B; // is the same as "friend class B"
};
```

4.27.2 See also

- [--strict, --no_strict](#) on page 3-194.

4.28 Read/write constants

A linkage specification for external constants indicates that a constant can be dynamically initialized or have mutable members.

Note

The use of "C++:read/write" linkage is only necessary for code compiled with `--apcs /rwp`. If you recompile existing code with this option, you must change the linkage specification for external constants that are dynamically initialized or have mutable members.

Compiling C++ with the `--apcs /rwp` option deviates from the ISO C++ Standard. The declarations in [Example 4-1](#) assume that `x` is in a read-only segment.

Example 4-1 External access

```
extern const T x;
extern "C++" const T x;
extern "C" const T x;
```

Dynamic initialization of `x` including user-defined constructors is not possible for constants and `T` cannot contain mutable members. The new linkage specification in [Example 4-2](#) declares that `x` is in a read/write segment even if it is initialized with a constant. Dynamic initialization of `x` is permitted and `T` can contain mutable members. The definitions of `x`, `y`, and `z` in another file must have the same linkage specifications.

Example 4-2 Linkage specification

```
extern const int z;                /* in read-only segment, cannot */
                                   /* be dynamically initialized */
extern "C++:read/write" const int y; /* in read/write segment */
                                   /* can be dynamically initialized */
extern "C++:read/write"
{
    const int i=5;                /* placed in read-only segment, */
                                   /* not extern because implicitly static */
    extern const T x=6;           /* placed in read/write segment */
    struct S
    {
        static const T T x;       /* placed in read/write segment */
    };
}
```

Constant objects must not be redeclared with another linkage. The code in [Example 4-3](#) produces a compile error.

Example 4-3 Compiler error

```
extern "C++" const T x;
extern "C++:read/write" const T x; /* error */
```

Note

Because C does not have the linkage specifications, you cannot use a **const** object declared in C++ as `extern "C++:read/write"` from C.

4.28.1 See also

- [--apcs=qualifer...qualifier](#) on page 3-11.

4.29 Scalar type constants

Constants of scalar type can be defined within classes. This is an old form. The modern form uses an initialized static data member.

4.29.1 Errors

A warning is issued if you define a member of constant integral type within a class.

4.29.2 Example

```
class A
{
    const int size = 10; // must be static
    int a[size];
};
```

4.30 Specialization of nonmember function templates

As an extension, it is permitted to specify a storage class on a specialization of a nonmember function template.

4.31 Type conversions

Type conversion between a pointer to an extern "C" function and a pointer to an extern "C++" function is permitted.

4.31.1 Example

```
extern "C" void f(); // f's type has extern "C" linkage
void (*pf)() = &f; // pf points to an extern "C++" function
// error unless implicit conversion is allowed
```

4.32 Standard C and Standard C++ language extensions

The compiler supports numerous extensions to both the ISO C99 and the ISO C++ Standards, such as various integral type extensions, various floating-point extensions, hexadecimal floating-point constants, and anonymous classes, structures, and unions.

These extensions are available if:

- the source language is C++ and you are compiling in nonstrict mode
- the source language is C99 and you are compiling in nonstrict mode
- the source language is C90 and you are compiling in nonstrict mode.

These extensions are not available if:

- the source language is C++ and the compiler is restricted to compiling strict C++ using the `--strict` compiler option.
- the source language is C99 and the compiler is restricted to compiling strict Standard C using the `--strict` compiler option.
- the source language is C90 and the compiler is restricted to compiling strict C90 using the `--strict` compiler option.

See:

- [Address of a register variable on page 4-36](#)
- [Arguments to functions on page 4-37](#)
- [Anonymous classes, structures and unions on page 4-38](#)
- [Assembler labels on page 4-39](#)
- [Empty declaration on page 4-40](#)
- [Hexadecimal floating-point constants on page 4-41](#)
- [Incomplete enums on page 4-42](#)
- [Integral type extensions on page 4-43](#)
- [Label definitions on page 4-44](#)
- [Long float on page 4-45](#)
- [Nonstatic local variables on page 4-46](#)
- [Structure, union, enum, and bitfield extensions on page 4-47.](#)

4.33 Address of a register variable

The address of a variable with **register** storage class can be taken.

4.33.1 Errors

The compiler generates a warning if you take the address of a variable with **register** storage class.

4.33.2 Example

```
void foo(void)
{
    register int i;
    int *j = &i;
}
```

4.34 Arguments to functions

Default arguments can be specified for function parameters other than those of a top-level function declaration. For example, they are accepted on typedef declarations and on pointer-to-function and pointer-to-member-function declarations.

4.35 Anonymous classes, structures and unions

Anonymous classes, structures, and unions are supported as an extension. Anonymous structures and unions are supported in C and C++.

Anonymous unions are available by default in C++. However, you must specify the `anon_unions` pragma if you want to use:

- anonymous unions and structures in C
- anonymous classes and structures in C++.

An anonymous union can be introduced into a containing class by a **typedef** name. Unlike a true anonymous union, it does not have to be declared directly. For example:

```
typedef union
{
    int i, j;
} U;           // U identifies a reusable anonymous union.
#pragma anon_unions
class A
{
    U;           // Okay -- references to A::i and A::j are allowed.
};
```

The extension also enables anonymous classes and anonymous structures, as long as they have no C++ features. For example, no static data members or member functions, no nonpublic members, and no nested types (except anonymous classes, structures, or unions) are allowed in anonymous classes and anonymous structures. For example:

```
#pragma anon_unions
struct A
{
    struct
    {
        int i, j;
    };           // Okay -- references to i and j
};           // through class A are allowed.

int foo(int m)
{
    A a;
    a.i = m;
    return a.i;
}
```

4.35.1 See also

- [#pragma anon_unions, #pragma no_anon_unions](#) on page 5-86.
- *Which GNU language extensions are supported by the ARM Compiler?*, <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka14717.html>

4.36 Assembler labels

Assembler labels specify the assembler name to use for a C symbol. For example, you might have assembler code and C code that uses the same symbol name, such as `counter`. Therefore, you can export a different name to be used by the assembler:

```
int counter __asm__("counter_v1") = 0;
```

This exports the symbol `counter_v1` and not the symbol `counter`.

4.36.1 See also

- [__asm](#) on page 5-9.

4.37 Empty declaration

An empty declaration, that is a semicolon with nothing before it, is permitted.

4.37.1 Example

```
; // do nothing
```

4.38 Hexadecimal floating-point constants

The ARM compiler implements an extension to the syntax of numeric constants in C to enable explicit specification of floating-point constants as IEEE bit patterns.

4.38.1 Syntax

The syntax for specifying floating-point constants as IEEE bit patterns is:

- `0f_n` Interpret an 8-digit hex number *n* as a **float** constant. There must be exactly eight digits.
- `0d_nn` Interpret a 16-digit hex number *nn* as a **double** constant. There must be exactly 16 digits.

4.39 Incomplete enums

Forward declarations of enums are supported.

4.39.1 Example

```
enum Incomplete_Enums_0;  
int Incomplete_Enums_2 (enum Incomplete_Enums_0 * passon)  
{  
    return 0;  
}  
int Incomplete_Enums_1 (enum Incomplete_Enums_0 * passon)  
{  
    return Incomplete_Enums_2(passon);  
}  
enum Incomplete_Enums_0 { ALPHA, BETA, GAMMA };
```

4.40 Integral type extensions

In an integral constant expression, an integral constant can be cast to a pointer type and then back to an integral type.

4.41 Label definitions

In Standard C and Standard C++, a statement must follow a label definition. In C and C++, a label definition can be followed immediately by a right brace.

4.41.1 Errors

The compiler generates a warning if a label definition is followed immediately by a right brace.

4.41.2 Example

```
void foo(char *p)
{
    if (p)
    {
        /* ... */
label:
    }
}
```

4.42 Long float

long float is accepted as a synonym for **double**.

4.43 Nonstatic local variables

Nonstatic local variables of an enclosing function can be referenced in a non-evaluated expression, for example, a `sizeof` expression inside a local class. A warning is issued.

4.44 Structure, union, enum, and bitfield extensions

The following structure, union, enum, and bitfield extensions are supported:

- In C, the element type of a file-scope array can be an incomplete **struct** or **union** type. The element type must be completed before its size is needed, for example, if the array is subscripted. If the array is not **extern**, the element type must be completed by the end of the compilation.
- The final semicolon preceding the closing brace } of a **struct** or **union** specifier can be omitted. A warning is issued.
- An initializer expression that is a single value and is used to initialize an entire static array, **struct**, or **union**, does not have to be enclosed in braces. ISO C requires the braces.
- An extension is supported to enable constructs similar to C++ anonymous unions, including the following:
 - not only anonymous unions but also anonymous structs are permitted. The members of anonymous structs are promoted to the scope of the containing **struct** and looked up like ordinary members.
 - they can be introduced into the containing **struct** by a **typedef** name. That is, they do not have to be declared directly, as is the case with true anonymous unions.
 - a tag can be declared but only in C mode.

To enable support for anonymous structures and unions, you must use the `anon_unions` pragma.

- An extra comma is permitted at the end of an **enum** list but a remark is issued.
- **enum** tags can be incomplete. You can define the tag name and resolve it later, by specifying the brace-enclosed list.
- The values of enumeration constants can be given by expressions that evaluate to unsigned quantities that fit in the **unsigned int** range but not in the **int** range. For example:


```
/* When ints are 32 bits: */
enum a { w = -2147483648 }; /* No error */
enum b { x = 0x80000000 }; /* No error */
enum c { y = 0x80000001 }; /* No error */
enum d { z = 2147483649 }; /* Error */
```
- Bit fields can have base types that are **enum** types or integral types besides **int** and **unsigned int**.

4.44.1 See also

- [Pragmas on page 5-85](#)
- [Structure, union, enum, and bitfield extensions](#)
- [New language features of C99 on page 6-85](#) in *Using the Compiler*.

4.45 GNU extensions to the C and C++ languages

GNU provides many extensions to the C and C++ languages and many of these extensions are supported by the ARM compiler. In GNU mode, all the GNU extensions to the relevant source language are available. Some GNU extensions are also available when you compile in a nonstrict mode.

To compile in GNU mode, use `--gnu`.

The following Standard C99 features are supported as GNU extensions in C90 and C++ when GNU mode is enabled:

- compound literals
- designated initializers
- elements of an aggregate initializer for an automatic variable are not required to be constant expressions

See *Which GNU language extensions are supported by the ARM Compiler*, <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka14717.html>.

The `asm` keyword is a Standard C++ feature that is supported as a GNU extension in C90 when GNU mode is enabled.

The following features are not part of any ISO standard but are supported as GNU extensions in either C90, C99, or C++ modes, when GNU mode is enabled:

- alternate keywords (C90, C99, C++)
- case ranges (C90, C99, C++)
- character escape sequence `'\e'` for escape character `<ESC>` (ASCII 27), (C90, C99, C++)
- dollar signs in identifiers (C90, C99, C++)
- labels as values (C90, C99 and C++)
- omission of middle operand in conditional statement if result is to be same as the test (C90, C99, C++)
- pointer arithmetic on `void` pointers and function pointers (C90 and C99 only)
- statement expressions (C90, C99 and C++)
- union casts (C90 and C99 only)
- unnamed fields in embedded structures and unions (C90, C99 and C++)
- zero-length arrays (C90 and C99 only).

4.45.1 See also

- [Appendix C Summary Table of GNU Language Extensions](#).

Chapter 5

Compiler-specific Features

This chapter describes the compiler-specific features:

- *Keywords and operators* on page 5-5
 - *__align* on page 5-6
 - *__alignof__* on page 5-7
 - *__ALIGNOF__* on page 5-8
 - *__asm* on page 5-9
 - *__forceinline* on page 5-10
 - *__global_reg* on page 5-11
 - *__inline* on page 5-13
 - *__int64* on page 5-14
 - *__INTADDR__* on page 5-15
 - *__irq* on page 5-16
 - *__packed* on page 5-17
 - *__pure* on page 5-20
 - *__smc* on page 5-21
 - *__softfp* on page 5-22
 - *__svc* on page 5-23
 - *__svc_indirect* on page 5-24
 - *__svc_indirect_r7* on page 5-25
 - *__value_in_regs* on page 5-26
 - *__weak* on page 5-27
 - *__writeonly* on page 5-29

- [__declspec attributes on page 5-30](#)
 - [__declspec\(dllexport\) on page 5-31](#)
 - [__declspec\(dllimport\) on page 5-33](#)
 - [__declspec\(noinline\) on page 5-34](#)
 - [__declspec\(noreturn\) on page 5-35](#)
 - [__declspec\(nothrow\) on page 5-36](#)
 - [__declspec\(notshared\) on page 5-37](#)
 - [__declspec\(thread\) on page 5-38](#)
- [Function attributes on page 5-39](#)
 - [__attribute__\(\(alias\)\) function attribute on page 5-41](#)
 - [__attribute__\(\(always_inline\)\) function attribute on page 5-42](#)
 - [__attribute__\(\(const\)\) function attribute on page 5-43](#)
 - [__attribute__\(\(constructor\[*priority*\]\)\) function attribute on page 5-44](#)
 - [__attribute__\(\(deprecated\)\) function attribute on page 5-46](#)
 - [__attribute__\(\(destructor\[*priority*\]\)\) function attribute on page 5-47](#)
 - [__attribute__\(\(format_arg\(*string-index*\)\)\) function attribute on page 5-48](#)
 - [__attribute__\(\(malloc\)\) function attribute on page 5-49](#)
 - [__attribute__\(\(noinline\)\) function attribute on page 5-50](#)
 - [__attribute__\(\(no_instrument_function\)\) function attribute on page 5-51](#)
 - [__attribute__\(\(nomerge\)\) function attribute on page 5-52](#)
 - [__attribute__\(\(nonnull\)\) function attribute on page 5-53](#)
 - [__attribute__\(\(noreturn\)\) function attribute on page 5-54](#)
 - [__attribute__\(\(notailcall\)\) function attribute on page 5-55](#)
 - [__attribute__\(\(pcs\("calling_convention"\)\)\) on page 5-56](#)
 - [__attribute__\(\(pure\)\) function attribute on page 5-57](#)
 - [__attribute__\(\(section\("name"\)\)\) function attribute on page 5-58](#)
 - [__attribute__\(\(unused\)\) function attribute on page 5-59](#)
 - [__attribute__\(\(used\)\) function attribute on page 5-60](#)
 - [__attribute__\(\(visibility\("visibility_type"\)\)\) function attribute on page 5-61](#)
 - [__attribute__\(\(weak\)\) function attribute on page 5-62](#)
 - [__attribute__\(\(weakref\("target"\)\)\) function attribute on page 5-63](#)
- [Type attributes on page 5-64](#)
 - [__attribute__\(\(bitband\)\) type attribute on page 5-65](#)
 - [__attribute__\(\(aligned\)\) type attribute on page 5-67](#)
 - [__attribute__\(\(packed\)\) type attribute on page 5-68](#)
 - [__attribute__\(\(transparent_union\)\) type attribute on page 5-69](#)
- [Variable attributes on page 5-70](#)
 - [__attribute__\(\(alias\)\) variable attribute on page 5-71](#)
 - [__attribute__\(\(at\(address\)\)\) variable attribute on page 5-72](#)
 - [__attribute__\(\(aligned\)\) variable attribute on page 5-73](#)
 - [__attribute__\(\(deprecated\)\) variable attribute on page 5-74](#)
 - [__attribute__\(\(noinline\)\) constant variable attribute on page 5-75](#)
 - [__attribute__\(\(packed\)\) variable attribute on page 5-76](#)
 - [__attribute__\(\(section\("name"\)\)\) variable attribute on page 5-77](#)
 - [__attribute__\(\(transparent_union\)\) variable attribute on page 5-78](#)
 - [__attribute__\(\(unused\)\) variable attribute on page 5-79](#)
 - [__attribute__\(\(used\)\) variable attribute on page 5-80](#)

- `__attribute__((visibility("visibility_type")))` variable attribute on page 5-81
- `__attribute__((weak))` variable attribute on page 5-82
- `__attribute__((weakref("target")))` variable attribute on page 5-83
- `__attribute__((zero_init))` variable attribute on page 5-84
- **Pragmas** on page 5-85
 - `#pragma anon_unions, #pragma no_anon_unions` on page 5-86
 - `#pragma arm` on page 5-87
 - `#pragma arm section [section_type_list]` on page 5-88
 - `#pragma diag_default tag[,tag,...]` on page 5-90
 - `#pragma diag_error tag[,tag,...]` on page 5-91
 - `#pragma diag_remark tag[,tag,...]` on page 5-92
 - `#pragma diag_suppress tag[,tag,...]` on page 5-93
 - `#pragma diag_warning tag[, tag, ...]` on page 5-94
 - `#pragma exceptions_unwind, #pragma no_exceptions_unwind` on page 5-95
 - `#pragma GCC system_header` on page 5-96
 - `#pragma hdrstop` on page 5-97
 - `#pragma hdrstop` on page 5-97
 - `#pragma import symbol_name` on page 5-98
 - `#pragma import(__use_full_stdio)` on page 5-99
 - `#pragma import(__use_smaller_memcpy)` on page 5-100
 - `#pragma inline, #pragma no_inline` on page 5-101
 - `#pragma no_pch` on page 5-102
 - `#pragma Onum` on page 5-103
 - `#pragma once` on page 5-104
 - `#pragma Ospace` on page 5-105
 - `#pragma Otime` on page 5-106
 - `#pragma pack(n)` on page 5-107
 - `#pragma pop` on page 5-109
 - `#pragma push` on page 5-110
 - `#pragma softfp_linkage, #pragma no_softfp_linkage` on page 5-111
 - `#pragma thumb` on page 5-112
 - `#pragma unroll [(n)]` on page 5-113
 - `#pragma unroll_completely` on page 5-115
 - `#pragma weak symbol, #pragma weak symbol1 = symbol2` on page 5-116
- **Instruction intrinsics** on page 5-117
 - `__breakpoint intrinsic` on page 5-118
 - `__cdp intrinsic` on page 5-119
 - `__clrex intrinsic` on page 5-120
 - `__clz intrinsic` on page 5-121
 - `__current_pc intrinsic` on page 5-122
 - `__current_sp intrinsic` on page 5-123
 - `__disable_fiq intrinsic` on page 5-124
 - `__disable_irq intrinsic` on page 5-125
 - `__enable_fiq intrinsic` on page 5-127
 - `__enable_irq intrinsic` on page 5-128
 - `__fabs intrinsic` on page 5-129
 - `__fabsf intrinsic` on page 5-130

- [__force_stores intrinsic](#) on page 5-131
- [__ldrex intrinsic](#) on page 5-132
- [__ldrexld intrinsic](#) on page 5-134
- [__ldrt intrinsic](#) on page 5-135
- [__memory_changed intrinsic](#) on page 5-137
- [__nop intrinsic](#) on page 5-138
- [__pld intrinsic](#) on page 5-141
- [__pldw intrinsic](#) on page 5-142
- [__pli intrinsic](#) on page 5-143
- [__promise intrinsic](#) on page 5-144
- [__qadd intrinsic](#) on page 5-145
- [__qdbl intrinsic](#) on page 5-146
- [__qsub intrinsic](#) on page 5-147
- [__rbit intrinsic](#) on page 5-148
- [__rev intrinsic](#) on page 5-149
- [__return_address intrinsic](#) on page 5-150
- [__ror intrinsic](#) on page 5-151
- [__schedule_barrier intrinsic](#) on page 5-152
- [__semihost intrinsic](#) on page 5-153
- [__sev intrinsic](#) on page 5-155
- [__sqrt intrinsic](#) on page 5-156
- [__sqrtf intrinsic](#) on page 5-157
- [__ssat intrinsic](#) on page 5-158
- [__strex intrinsic](#) on page 5-159
- [__strexld intrinsic](#) on page 5-161
- [__strt intrinsic](#) on page 5-162
- [__swp intrinsic](#) on page 5-163
- [__usat intrinsic](#) on page 5-164
- [__wfe intrinsic](#) on page 5-165
- [__wfi intrinsic](#) on page 5-166
- [__yield intrinsic](#) on page 5-167
- [ARMv6 SIMD intrinsics](#) on page 5-168
- [ETSI basic operations](#) on page 5-169
- [C55x intrinsics](#) on page 5-171
- [VFP status intrinsic](#) on page 5-173
 - [__vfp_status intrinsic](#) on page 5-174
- [Fused Multiply Add \(FMA\) intrinsics](#) on page 5-175
- [Named register variables](#) on page 5-176
- [GNU built-in functions](#) on page 5-180
- [Predefined macros](#) on page 5-183
- [Built-in function name variables](#) on page 5-189.

5.1 Keywords and operators

This topic lists the function keywords and operators supported by the compiler armcc.

[Table 5-1](#) lists keywords that are ARM extensions to the C and C++ Standards. Standard C and Standard C++ keywords that do not have behavior or restrictions specific to the ARM compiler are not documented in the table.

Table 5-1 Keyword extensions supported by the ARM compiler

Keywords		
<code>__align</code>	<code>__int64</code>	<code>__svc</code>
<code>__ALIGNOF__</code>	<code>__INTADDR__</code>	<code>__svc_indirect</code>
<code>__asm</code>	<code>__irq</code>	<code>__svc_indirect_r7</code>
<code>__declspec</code>	<code>__packed</code>	<code>__value_in_regs</code>
<code>__forceinline</code>	<code>__pure</code>	<code>__weak</code>
<code>__global_reg</code>	<code>__softfp</code>	<code>__writeonly</code>
<code>__inline</code>	<code>__smc</code>	

5.2 `__align`

The `__align` keyword instructs the compiler to align a variable on an n -byte boundary. `__align` is a storage class modifier. It does not affect the type of the function.

5.2.1 Syntax

```
__align( $n$ )
```

Where:

n is the alignment boundary.

For local variables, n can take the values 1, 2, 4, or 8.

For global variables, n can take any value up to 0x80000000 in powers of 2.

5.2.2 Usage

`__align(n)` is useful when the normal alignment of the variable being declared is less than n . Eight-byte alignment can give a significant performance advantage with VFP instructions.

`__align` can be used in conjunction with **extern** and **static**.

5.2.3 Restrictions

Because `__align` is a storage class modifier, it cannot be used on:

- types, including **typedefs** and structure definitions
- function parameters.

You can only overalign. That is, you can make a two-byte object four-byte aligned but you cannot align a four-byte object at 2 bytes.

5.2.4 Examples

```
__align(8) char buffer[128]; // buffer starts on eight-byte boundary

void foo(void)
{
    ...
    __align(16) int i; // this alignment value is not permitted for
                      // a local variable
    ...
}

__align(16) int i; // permitted as a global variable.
```

5.2.5 See also

- [--min_array_alignment=opt](#) on page 3-147 in *Using the Compiler*.

5.3 `__alignof__`

The `__alignof__` keyword enables you to enquire about the alignment of a type or variable.

———— **Note** ————

This keyword is a GNU compiler extension that the ARM compiler supports.

5.3.1 Syntax

```
__alignof__(type)
```

```
__alignof__(expr)
```

Where:

`type` is a type

`expr` is an lvalue.

5.3.2 Return value

`__alignof__(type)` returns the alignment requirement for the type `type`, or 1 if there is no alignment requirement.

`__alignof__(expr)` returns the alignment requirement for the type of the lvalue `expr`, or 1 if there is no alignment requirement.

5.3.3 Example

```
int Alignment_0(void)
{
    return __alignof__(int);
}
```

5.3.4 See also

- [__ALIGNOF__](#) on page 5-8.

5.4 `__ALIGNOF__`

The `__ALIGNOF__` keyword returns the alignment requirement for a specified type, or for the type of a specified object.

5.4.1 Syntax

```
__ALIGNOF__(type)
```

```
__ALIGNOF__(expr)
```

Where:

`type` is a type

`expr` is an lvalue.

5.4.2 Return value

`__ALIGNOF__(type)` returns the alignment requirement for the type `type`, or 1 if there is no alignment requirement.

`__ALIGNOF__(expr)` returns the alignment requirement for the type of the lvalue `expr`, or 1 if there is no alignment requirement. The lvalue itself is not evaluated.

5.4.3 Example

```
typedef struct s_foo { int i; short j; } foo;
typedef __packed struct s_bar { int i; short j; } bar;
return __ALIGNOF(struct s_foo); // returns 4
return __ALIGNOF(foo);         // returns 4
return __ALIGNOF(bar);         // returns 1
```

5.4.4 See also

- [__alignof__](#) on page 5-7.

5.5 `__asm`

This keyword is used to pass information from the compiler to the ARM assembler `armasm`. The precise action of this keyword depends on its usage.

5.5.1 Usage

Embedded assembler

The `__asm` keyword can be used to declare or define an embedded assembly function. For example:

```
__asm void my_strcpy(const char *src, char *dst);
```

See *Embedded assembler support in the compiler on page 8-36* in *Using the Compiler* for more information.

Inline assembler

The `__asm` keyword can be used to incorporate inline assembly into a function. For example:

```
int qadd(int i, int j)
{
    int res;
    __asm
    {
        QADD    res, i, j
    }
    return res;
}
```

See *Compiler support for inline assembly language on page 8-4* in *Using the Compiler* for more information.

Assembler labels

The `__asm` keyword can be used to specify an assembler label for a C symbol. For example:

```
int count __asm__("count_v1"); // export count_v1, not count
```

See *Assembler labels on page 4-39* for more information.

Named register variables

The `__asm` keyword can be used to declare a named register variable. For example:

```
register int foo __asm("r0");
```

See *Named register variables on page 5-176* for more information.

5.6 `__forceinline`

The `__forceinline` keyword forces the compiler to compile a C or C++ function inline.

The semantics of `__forceinline` are exactly the same as those of the C++ `inline` keyword. The compiler attempts to inline a function qualified as `__forceinline`, regardless of its characteristics. However, the compiler does not inline a function if doing so causes problems. For example, a recursive function is inlined into itself only once.

`__forceinline` is a storage class qualifier. It does not affect the type of a function.

———— **Note** —————

This keyword has the function attribute equivalent `__attribute__((always_inline))`.

5.6.1 Example

```
__forceinline static int max(int x, in y)
{
    return x > y ? x : y; // always inline if possible
}
```

5.6.2 See also

- [--forceinline on page 3-95](#)
- [__attribute__\(\(always_inline\)\) function attribute on page 5-42.](#)

5.7 `__global_reg`

The `__global_reg` storage class specifier allocates the declared variable to a global variable register.

5.7.1 Syntax

```
__global_reg(n) type varName
```

Where:

- | | |
|----------------------|--|
| <code>n</code> | Is an integer between one and eight. |
| <code>type</code> | Is one of the following types: <ul style="list-style-type: none"> • any integer type, except long long • any char type • any pointer type. |
| <code>varName</code> | Is the name of a variable. |

5.7.2 Restrictions

If you use this storage class, you cannot use any additional storage class such as **extern**, **static**, or **typedef**.

In C, global register variables cannot be qualified or initialized at declaration. In C++, any initialization is treated as a dynamic initialization.

The number of available registers varies depending on the variant of the AAPCS being used, there are between five and seven registers available for use as global variable registers.

In practice, it is recommended that you do not use more than:

- three global register variables in ARM or Thumb on a processor with Thumb-2 technology
- one global register variable in Thumb on a processor without Thumb-2 technology
- half the number of available floating-point registers as global floating-point register variables.

If you declare too many global variables, code size increases significantly. In some cases, your program might not compile.

———— **Caution** ————

You must take care when using global register variables because:

- There is no check at link time to ensure that direct calls between different compilation units are sensible. If possible, define global register variables used in a program in each compilation unit of the program. In general, it is best to place the definition in a global header file. You must set up the value in the global register early in your code, before the register is used.
- A global register variable maps to a callee-saved register, so its value is saved and restored across a call to a function in a compilation unit that does not use it as a global register variable, such as a library function.

- Calls back into a compilation unit that uses a global register variable are dangerous. For example, if a function using a global register is called from a compilation unit that does not declare the global register variable, the function reads the wrong values from its supposed global register variables.
 - This storage class can only be used at file scope.
 - Volatile variables with the `__global_reg` storage class specifier are not treated as volatile.
-

5.7.3 Example

[Example 5-1](#) declares a global variable register allocated to r5.

Example 5-1 Declaring a global integer register variable

```
__global_reg(2) int x; v2 is the synonym for r5
```

[Example 5-2](#) produces an error because global registers must be specified in all declarations of the same variable.

Example 5-2 Global register - declaration error

```
int x;
__global_reg(1) int x; // error
```

In C, `__global_reg` variables cannot be initialized at definition. [Example 5-3](#) produces an error in C, but not in C++.

Example 5-3 Global register - initialization error

```
__global_reg(1) int x=1; // error in C, OK in C++
```

5.7.4 See also

- [--global_reg=reg_name\[,reg_name,...\]](#) on page 3-106.

5.8 `__inline`

The `__inline` keyword suggests to the compiler that it compiles a C or C++ function inline, if it is sensible to do so.

The semantics of `__inline` are exactly the same as those of the `inline` keyword. However, `inline` is not available in C90.

`__inline` is a storage class qualifier. It does not affect the type of a function.

5.8.1 Example

```
__inline int f(int x)
{
    return x*5+1;
}
int g(int x, int y)
{
    return f(x) + f(y);
}
```

5.8.2 See also

- [Inline functions on page 6-30](#) in *Using the Compiler*.

5.9 `__int64`

The `__int64` keyword is a synonym for the keyword sequence `long long`.
`__int64` is accepted even when using `--strict`.

5.9.1 See also

- [--strict, --no_strict on page 3-194](#)
- [long long on page 4-14](#).

5.10 `__INTADDR__`

The `__INTADDR__` operation treats the enclosed expression as a constant expression, and converts it to an integer constant.

———— **Note** —————

This is used in the `offsetof` macro.

5.10.1 Syntax

`__INTADDR(expr)`

Where:

expr is an integral constant expression.

5.10.2 Return value

`__INTADDR__(expr)` returns an integer constant equivalent to *expr*.

5.10.3 See also

- [Restrictions on embedded assembly language functions in C and C++ code on page 8-40 in *Using the Compiler*.](#)

5.11 `__irq`

The `__irq` keyword enables a C or C++ function to be used as an interrupt routine.

`__irq` is a function qualifier. It affects the type of the function.

5.11.1 Restrictions

All corrupted registers except floating-point registers are preserved, not only those that are normally preserved under the AAPCS. The default AAPCS mode must be used.

No arguments or return values can be used with `__irq` functions.

In architectures other than ARMv6-M and ARMv7-M, the function exits by setting the program counter to `1r-4` and the CPSR to the value in `SPSR`.

Note

In ARMv6-M and ARMv7-M, specifying `__irq` does not affect the behavior of the compiled output. However, ARM recommends using `__irq` on exception handlers for clarity and easier software porting.

Note

- For architectures that support ARM and Thumb-2 technology, for example ARMv6T2, ARMv7-A, and ARMv7-R, functions specified as `__irq` compile to ARM or Thumb code depending on whether the compile option or `#pragma` specify ARM or Thumb.
 - For Thumb only architectures, for example ARMv6-M and ARMv7-M, functions specified as `__irq` compile to Thumb code.
 - For architectures before ARMv6T2, functions specified as `__irq` compile to ARM code even if you compile with `--thumb` or `#pragma thumb`.
-

5.11.2 See also

- [--thumb](#) on page 3-197
- [--arm](#) on page 3-15
- [#pragma thumb](#) on page 5-112
- [#pragma arm](#) on page 5-87
- [ARM, Thumb, and ThumbEE instruction sets](#) on page 3-3 in *Using the Assembler*
- [Chapter 6 Handling Processor Exceptions](#) in *Developing Software for ARM Processors*.

5.12 `__packed`

The `__packed` qualifier sets the alignment of any valid type to 1. This means that:

- there is no padding inserted to align the packed object
- objects of packed type are read or written using unaligned accesses.

The `__packed` qualifier applies to all members of a structure or union when it is declared using `__packed`. There is no padding between members, or at the end of the structure. All substructures of a packed structure must be declared using `__packed`. Integral subfields of an unpacked structure can be packed individually.

5.12.1 Usage

The `__packed` qualifier is useful to map a structure to an external data structure, or for accessing unaligned data, but it is generally not useful to save data size because of the relatively high cost of unaligned access. Only packing fields in a structure that requires packing can reduce the number of unaligned accesses.

———— **Note** —————

On ARM processors that do not support unaligned access in hardware, for example, pre-ARMv6, access to unaligned data can be costly in terms of code size and execution speed. Data accesses through packed structures must be minimized to avoid increase in code size and performance loss.

5.12.2 Restrictions

The following restrictions apply to the use of `__packed`:

- The `__packed` qualifier cannot be used on structures that were previously declared without `__packed`.
- Unlike other type qualifiers you cannot have both a `__packed` and non-`__packed` version of the same structure type.
- The `__packed` qualifier does not affect local variables of integral type.
- A packed structure or union is not assignment-compatible with the corresponding unpacked structure. Because the structures have a different memory layout, the only way to assign a packed structure to an unpacked structure is by a field-by-field copy.
- The effect of casting away `__packed` is undefined, except on `char` types. The effect of casting a nonpacked structure to a packed structure, or a packed structure to a nonpacked structure, is undefined. A pointer to an integral type that is not packed can be legally cast, explicitly or implicitly, to a pointer to a packed integral type.
- There are no packed array types. A packed array is an array of objects of packed type. There is no padding in the array.

5.12.3 Errors

Taking the address of a field in a `__packed` structure or a `__packed`-qualified field yields a `__packed`-qualified pointer. The compiler will produce a type error if you attempt to implicitly cast this pointer to a non-`__packed` pointer. This contrasts with its behavior for address-taken fields of a `#pragma packed` structure.

5.12.4 Example

Example 5-4 shows that a pointer can point to a packed type.

Example 5-4 Pointer to packed

```
typedef __packed int* PpI;      /* pointer to a __packed int */
__packed int *p;              /* pointer to a __packed int */
PpI p2;                       /* 'p2' has the same type as 'p' */
                               /* __packed is a qualifier */
                               /* like 'const' or 'volatile' */
typedef int *PI;              /* pointer to int */
__packed PI p3;               /* a __packed pointer to a normal int */
                               /* -- not the same type as 'p' and 'p2' */
int *__packed p4;             /* 'p4' has the same type as 'p3' */
```

Example 5-5 shows that when a packed object is accessed using a pointer, the compiler generates code that works and that is independent of the pointer alignment.

Example 5-5 Packed structure

```
typedef __packed struct
{
    char x;                    // all fields inherit the __packed qualifier
    int y;
} X;                           // 5 byte structure, natural alignment = 1
int f(X *p)
{
    return p->y;               // does an unaligned read
}
typedef struct
{
    short x;
    char y;
    __packed int z;           // only pack this field
    char a;
} Y;                           // 8 byte structure, natural alignment = 2
int g(Y *p)
{
    return p->z + p->x;         // only unaligned read for z
}
```

5.12.5 See also

- [__attribute__\(\(packed\)\) type attribute on page 5-68](#)
- [__attribute__\(\(packed\)\) variable attribute on page 5-76](#)
- [#pragma pack\(n\) on page 5-107](#)
- [Packed structures on page 6-11](#)
- [The __packed qualifier and unaligned data access in C and C++ code on page 6-47 in Using the Compiler](#)

- *Comparisons of an unpacked struct, a `__packed` struct, and a struct with individually `__packed` fields, and of a `__packed` struct and a `#pragma packed` struct on page 6-52 in *Using the Compiler*.*

5.13 `__pure`

The `__pure` keyword asserts that a function declaration is pure.

A function is *pure* only if:

- the result depends exclusively on the values of its arguments
- the function has no side effects.

`__pure` is a function qualifier. It affects the type of a function.

———— **Note** —————

This keyword has the function attribute equivalent `__attribute__((const))`.

5.13.1 Default

By default, functions are assumed to be impure.

5.13.2 Usage

Pure functions are candidates for common subexpression elimination.

5.13.3 Restrictions

A function that is declared as pure can have no side effects. For example, pure functions:

- cannot call impure functions
- cannot use global variables or dereference pointers, because the compiler assumes that the function does not access memory, except stack memory
- must return the same value each time when called twice with the same parameters.

5.13.4 Example

```
int factr(int n) __pure
{
    int f = 1;
    while (n > 0)
        f *= n--;
    return f;
}
```

5.13.5 See also

- [__attribute__\(\(const\)\) function attribute](#) on page 5-43
- [Functions that return the same result when called with the same arguments](#) on page 6-25 in *Using the Compiler*
- [Recommendation of postfix syntax when qualifying functions with ARM function modifiers](#) on page 6-28 in *Using the Compiler*.

5.14 __smc

The `__smc` keyword declares an SMC (*Secure Monitor Call*) function. A call to the SMC function inserts an SMC instruction into the instruction stream generated by the compiler at the point of function invocation.

———— **Note** ————

The SMC instruction replaces the SMI instruction used in previous versions of the ARM assembly language.

`__smc` is a function qualifier. It affects the type of a function.

5.14.1 Syntax

```
__smc(int smc_num) return-type function-name([argument-list]);
```

Where:

smc_num Is a 4-bit immediate value used in the SMC instruction.
The value of *smc_num* is ignored by the ARM processor, but can be used by the SMC exception handler to determine what service is being requested.

5.14.2 Restrictions

The SMC instruction is available for selected ARM architecture-based processors, if they have the Security Extensions. See [SMC on page 3-132](#) in the *Assembler Reference* for more information.

The compiler generates an error if you compile source code containing the `__smc` keyword for an architecture that does not support the SMC instruction.

5.14.3 Example

```
__smc(5) void mycall(void); /* declare a name by which SMC #5 can be called */
...
mycall(); /* invoke the function */
```

5.14.4 See also

- [--cpu=name on page 3-49](#)
- [SMC on page 3-132](#) in the *Assembler Reference*.

5.15 `__softfp`

The `__softfp` keyword asserts that a function uses software floating-point linkage.

`__softfp` is a function qualifier. It affects the type of the function.

Note

This keyword has the `#pragma` equivalent `#pragma __softfp_linkage`.

5.15.1 Usage

Calls to the function pass floating-point arguments in integer registers. If the result is a floating-point value, the value is returned in integer registers. This duplicates the behavior of compilation targeting software floating-point.

This keyword enables the same library to be used by sources compiled to use hardware and software floating-point.

Note

In C++, if a virtual function qualified with the `__softfp` keyword is to be overridden, the overriding function must also be declared as `__softfp`. If the functions do not match, the compiler generates an error.

5.15.2 See also

- [__attribute__\(\(pcs\("calling_convention"\)\)\)](#) on page 5-56
- [--fpu=name](#) on page 3-100
- [#pragma softfp_linkage](#), [#pragma no_softfp_linkage](#) on page 5-111
- [Compiler support for floating-point computations and linkage](#) on page 6-65 in *Using the Compiler*.

5.16 __svc

The `__svc` keyword declares a *SuperVisor Call* (SVC) function taking up to four integer-like arguments and returning up to four results in a `value_in_regs` structure.

`__svc` is a function qualifier. It affects the type of a function.

5.16.1 Syntax

```
__svc(int svc_num) return-type function-name([argument-list]);
```

Where:

`svc_num` Is the immediate value used in the SVC instruction.
It is an expression evaluating to an integer in the range:

- 0 to $2^{24}-1$ (a 24-bit value) in an ARM instruction
- 0-255 (an 8-bit value) in a 16-bit Thumb instruction.

5.16.2 Usage

This causes function invocations to be compiled inline as an AAPCS-compliant operation that behaves similarly to a normal call to a function.

The `__value_in_regs` qualifier can be used to specify that a small structure of up to 16 bytes is returned in registers, rather than by the usual structure-passing mechanism defined in the AAPCS.

5.16.3 Example

```
__svc(42) void terminate_1(int procnum); // terminate_1 returns no results
__svc(42) int terminate_2(int procnum); // terminate_2 returns one result
typedef struct res_type
{
    int res_1;
    int res_2;
    int res_3;
    int res_4;
} res_type;
__svc(42) __value_in_regs res_type terminate_3(int procnum);
// terminate_3 returns more than
// one result
```

5.16.4 Errors

When an ARM architecture variant or ARM architecture-based processor that does not support an SVC instruction is specified on the command line using the `--cpu` option, the compiler generates an error.

5.16.5 See also

- [--cpu=name](#) on page 3-49
- [__value_in_regs](#) on page 5-26
- [SVC](#) on page 3-154 in the *Assembler Reference*.

5.17 __svc_indirect

The `__svc_indirect` keyword passes an operation code to the SVC handler in r12.

`__svc_indirect` is a function qualifier. It affects the type of a function.

5.17.1 Syntax

```
__svc_indirect(int svc_num)
    return-type function-name(int real_num[, argument-list]);
```

Where:

<i>svc_num</i>	Is the immediate value used in the SVC instruction. It is an expression evaluating to an integer in the range: <ul style="list-style-type: none"> • 0 to $2^{24}-1$ (a 24-bit value) in an ARM instruction • 0-255 (an 8-bit value) in a 16-bit Thumb instruction.
<i>real_num</i>	Is the value passed in r12 to the handler to determine the function to perform.

To use the indirect mechanism, your system handlers must make use of the r12 value to select the required operation.

5.17.2 Usage

You can use this feature to implement indirect SVCs.

5.17.3 Example

```
int __svc_indirect(0) ioctl(int svcino, int fn, void *argp);
```

Calling:

```
ioctl(IOCTL+4, RESET, NULL);
```

compiles to SVC #0 with IOCTL+4 in r12.

5.17.4 Errors

When an ARM architecture variant or ARM architecture-based processor that does not support an SVC instruction is specified on the command line using the `--cpu` option, the compiler generates an error.

5.17.5 See also

- [--cpu=name](#) on page 3-49
- [__value_in_regs](#) on page 5-26
- [SVC](#) on page 3-154 in the *Assembler Reference*.

5.18 __svc_indirect_r7

The `__svc_indirect_r7` keyword behaves like `__svc_indirect`, but uses `r7` instead of `r12`.
`__svc_indirect_r7` is a function qualifier. It affects the type of a function.

5.18.1 Syntax

```
__svc_indirect_r7(int svc_num)
    return-type function-name(int real_num[], argument-list);
```

Where:

svc_num Is the immediate value used in the SVC instruction.

It is an expression evaluating to an integer in the range:

- 0 to $2^{24}-1$ (a 24-bit value) in an ARM instruction
- 0-255 (an 8-bit value) in a 16-bit Thumb instruction.

real_num Is the value passed in `r7` to the handler to determine the function to perform.

5.18.2 Usage

Thumb applications on ARM Linux use `__svc_indirect_r7` to make kernel syscalls.

You can also use this feature to implement indirect SVCs.

5.18.3 Example

```
long __svc_indirect_r7(0) \
    SVC_write(unsigned, int fd, const char * buf, size_t count);
#define write(fd, buf, count) SVC_write(4, (fd), (buf), (count))
```

Calling:

```
write(fd, buf, count);
```

compiles to SVC #0 with `r0 = fd`, `r1 = buf`, `r2 = count`, and `r7 = 4`.

5.18.4 Errors

When an ARM architecture variant or ARM architecture-based processor that does not support an SVC instruction is specified on the command line using the `--cpu` option, the compiler generates an error.

5.18.5 See also

- [__value_in_regs](#) on page 5-26
- [--cpu=name](#) on page 3-49
- [SVC](#) on page 3-154 in the *Assembler Reference*.

5.19 __value_in_regs

The `__value_in_regs` qualifier instructs the compiler to return a structure of up to four integer words in integer registers or up to four floats or doubles in floating-point registers rather than using memory.

`__value_in_regs` is a function qualifier. It affects the type of a function.

5.19.1 Syntax

```
__value_in_regs return-type function-name([argument-list]);
```

Where:

`return-type` is the type of a structure of up to four words in size.

5.19.2 Usage

Declaring a function `__value_in_regs` can be useful when calling functions that return more than one result.

5.19.3 Restrictions

A C++ function cannot return a `__value_in_regs` structure if the structure requires copy constructing.

If a virtual function declared as `__value_in_regs` is to be overridden, the overriding function must also be declared as `__value_in_regs`. If the functions do not match, the compiler generates an error.

5.19.4 Errors

Where the structure returned in a function qualified by `__value_in_regs` is too big, a warning is produced and the `__value_in_regs` structure is then ignored.

5.19.5 Example

```
typedef struct int64_struct
{
    unsigned int lo;
    unsigned int hi;
} int64_struct;
__value_in_regs extern
int64_struct mul64(unsigned a, unsigned b);
```

5.19.6 See also

- [Functions that return multiple values through registers on page 6-24 in *Using the ARM Compiler*.](#)

5.20 `__weak`

This keyword instructs the compiler to export symbols weakly.

The `__weak` keyword can be applied to function and variable declarations, and to function definitions.

5.20.1 Usage

Functions and variable declarations

For declarations, this storage class specifies an **extern** object declaration that, even if not present, does not cause the linker to fault an unresolved reference.

For example:

```
__weak void f(void);
...
f(); // call f weakly
```

If the reference to a missing weak function is made from code that compiles to a branch or branch link instruction, then either:

- The reference is resolved as branching to the next instruction. This effectively makes the branch a NOP.
- The branch is replaced by a NOP instruction.

Function definitions

Functions defined with `__weak` export their symbols weakly. A weakly defined function behaves like a normally defined function unless a nonweakly defined function of the same name is linked into the same image. If both a nonweakly defined function and a weakly defined function exist in the same image then all calls to the function resolve to call the nonweak function. If multiple weak definitions are available, the linker generates an error message, unless the linker option `--muldefweak` is used. In this case, the linker chooses one for use by all calls.

Functions declared with `__weak` and then defined without `__weak` behave as nonweak functions.

5.20.2 Restrictions

There are restrictions when you qualify function and variable declarations, and function definitions, with `__weak`.

Functions and variable declarations

A function or variable cannot be used both weakly and nonweakly in the same compilation. For example, the following code uses `f()` weakly from `g()` and `h()`:

```
void f(void);
void g()
{
    f();
}
__weak void f(void);
void h()
{
    f();
}
```


It is not possible to use a function or variable weakly from the same compilation that defines the function or variable. The following code uses `f()` nonweakly from `h()`:

```
__weak void f(void);
void h()
{
    f();
}
void f() {}
```

The linker does not load the function or variable from a library unless another compilation uses the function or variable nonweakly. If the reference remains unresolved, its value is assumed to be NULL. Unresolved references, however, are not NULL if the reference is from code to a position-independent section or to a missing `__weak` function.

Function definitions

Weakly defined functions cannot be inlined.

5.20.3 Example

```
__weak const int c;           // assume 'c' is not present in final link
const int *f1() { return &c; } // '&c' returns non-NULL if
                               // compiled and linked /ropi
__weak int i;                // assume 'i' is not present in final link
int *f2() { return &i; }     // '&i' returns non-NULL if
                               // compiled and linked /rwp
__weak void f(void);         // assume 'f' is not present in final link
typedef void (*FP)(void);
FP g() { return f; }         // 'g' returns non-NULL if
                               // compiled and linked /ropi
```

5.20.4 See also

- *Creating Static Software Libraries with armar* for more information on library searching
- [--muldefweak, --no_muldefweak on page 2-113](#) in the *Linker Reference*.

5.21 __writeonly

The `__writeonly` type qualifier indicates that a data object cannot be read from.

In the C and C++ type system it behaves as a cv-qualifier like `const` or `volatile`. Its specific effect is that an lvalue with `__writeonly` type cannot be converted to an rvalue.

Assignment to a `__writeonly` bitfield is not allowed if the assignment is implemented as read-modify-write. This is implementation-dependent.

5.21.1 Example

```
void foo(__writeonly int *ptr)
{
    *ptr = 0; // allowed
    printf("ptr value = %d\n", *ptr); // error
}
```

5.22 __declspec attributes

The `__declspec` keyword enables you to specify special attributes of objects and functions. For example, you can use the `__declspec` keyword to declare imported or exported functions and variables, or to declare *Thread Local Storage* (TLS) objects.

The `__declspec` keyword must prefix the declaration specification. For example:

```
__declspec(noreturn) void overflow(void);
__declspec(thread) int i;
```

Table 5-2 summarizes the available `__declspec` attributes. `__declspec` attributes are storage class modifiers. They do not affect the type of a function or variable.

Table 5-2 __declspec attributes supported by the compiler and their equivalents

<code>__declspec</code> attribute	non <code>__declspec</code> equivalent
<code>__declspec(dllexport)</code>	-
<code>__declspec(dllimport)</code>	-
<code>__declspec(noinline)</code>	<code>__attribute__((noinline))</code> ^a
<code>__declspec(noreturn)</code>	<code>__attribute__((noreturn))</code> ^a
<code>__declspec(nothrow)</code>	-
<code>__declspec(notshared)</code>	-
<code>__declspec(thread)</code>	-

a. A GNU compiler extension supported by the ARM compiler.

5.23 `__declspec(dllexport)`

The `__declspec(dllexport)` attribute exports the definition of a symbol through the dynamic symbol table when building DLL libraries. On classes, it controls the visibility of class impedimenta such as vtables, construction vtables and RTTI, and sets the default visibility for member function and static data members.

5.23.1 Usage

You can use `__declspec(dllexport)` on a function, a class, or on individual members of a class.

When an inline function is marked `__declspec(dllexport)`, the function definition might be inlined, but an out-of-line instance of the function is always generated and exported in the same way as for a non-inline function.

When a class is marked `__declspec(dllexport)`, for example, `class __declspec(dllexport) S { ... }`; its static data members and member functions are all exported. When individual static data members and member functions are marked with `__declspec(dllexport)`, only those members are exported. vtables, construction vtable tables and RTTI are also exported.

———— Note ————

The following declaration is correct:

```
class __declspec(dllexport) S { ... };
```

The following declaration is incorrect:

```
__declspec(dllexport) class S { ... };
```

In conjunction with `--export_all_vtbl`, you can use `__declspec(notshared)` to exempt a class or structure from having its vtable, construction vtable table and RTTI exported. `--export_all_vtbl` and `__declspec(dllexport)` are typically not used together.

5.23.2 Restrictions

If you mark a class with `__declspec(dllexport)`, you cannot then mark individual members of that class with `__declspec(dllexport)`.

If you mark a class with `__declspec(dllexport)`, ensure that all of the base classes of that class are marked `__declspec(dllexport)`.

If you export a virtual function within a class, ensure that you either export all of the virtual functions in that class, or that you define them inline so that they are visible to the client.

5.23.3 Example

The `__declspec()` required in a declaration depends on whether or not the definition is in the same shared library.

```
/* This is the declaration for use in the same shared library as the */
/* definition */
__declspec(dllexport) extern int mymod_get_version(void);

/* Translation unit containing the definition */
__declspec(dllexport) extern int mymod_get_version(void)
{
    return 42;
}
```

```

/* This is the declaration for use in a shared library that does not contain */
/* the definition */
__declspec(dllimport) extern int mymod_get_version(void);

```

As a result of the following macro, a translation unit that does not have the definition in a defining link unit sees `__declspec(dllexport)`.

```

/* mymod.h - interface to my module */
#ifdef BUILDING_MYMOD
#define MYMOD_API __declspec(dllexport)
#else /* not BUILDING_MYMOD */
#define MYMOD_API __declspec(dllimport)
#endif

MYMOD_API int mymod_get_version(void);

```

5.23.4 See also

- [__declspec\(dllimport\)](#) on page 5-33
- [__declspec\(notshared\)](#) on page 5-37
- [--export_all_vtbl, --no_export_all_vtbl](#) on page 3-90
- [--use_definition_visibility](#) on page 2-177 in the *Linker Reference*
- [--visibility_inlines_hidden](#) on page 3-217.

5.24 `__declspec(dllimport)`

The `__declspec(dllimport)` attribute imports a symbol through the dynamic symbol table when linking against DLL libraries.

5.24.1 Usage

When an inline function is marked `__declspec(dllimport)`, the function definition in this compilation unit might be inlined, but is never generated out-of-line. An out-of-line call or address reference uses the imported symbol.

You can only use `__declspec(dllimport)` on **extern** functions and variables, and on classes.

When a class is marked `__declspec(dllimport)`, its static data members and member functions are all imported. When individual static data members and member functions are marked with `__declspec(dllimport)`, only those members are imported.

5.24.2 Restrictions

If you mark a class with `__declspec(dllimport)`, you cannot then mark individual members of that class with `__declspec(dllimport)`.

5.24.3 Examples

```
__declspec(dllimport) int i;
class __declspec(dllimport) X { void f(); };
```

5.24.4 See also

- [__declspec\(dllexport\)](#) on page 5-31.

5.25 `__declspec(noinline)`

The `__declspec(noinline)` attribute suppresses the inlining of a function at the call points of the function.

`__declspec(noinline)` can also be applied to constant data, to prevent the compiler from using the value for optimization purposes, without affecting its placement in the object. This is a feature that can be used for patchable constants, that is, data that is later patched to a different value. It is an error to try to use such constants in a context where a constant value is required. For example, an array dimension.

———— **Note** —————

This `__declspec` attribute has the function attribute equivalent `__attribute__((noinline))`.

5.25.1 Examples

```
/* Prevent y being used for optimization */
__declspec(noinline) const int y = 5;
/* Suppress inlining of foo() wherever foo() is called */
__declspec(noinline) int foo(void);
```

5.25.2 See also

- [#pragma inline, #pragma no_inline](#) on page 5-101
- [__attribute__\(\(noinline\)\) constant variable attribute](#) on page 5-75
- [__attribute__\(\(noinline\)\) function attribute](#) on page 5-50.

5.26 `__declspec(noreturn)`

The `__declspec(noreturn)` attribute asserts that a function never returns.

———— **Note** ————

This attribute has the function equivalent `__attribute__((noreturn))`. However, `__attribute__((noreturn))` and `__declspec(noreturn)` differ in that when compiling a function definition, if the function reaches an explicit or implicit return, `__attribute__((noreturn))` is ignored and the compiler generates a warning. This does not apply to `__declspec(noreturn)`.

5.26.1 Usage

Use this attribute to reduce the cost of calling a function that never returns, such as `exit()`. If a noreturn function returns to its caller, the behavior is undefined.

5.26.2 Restrictions

The return address is not preserved when calling the noreturn function. This limits the ability of a debugger to display the call stack.

5.26.3 Example

```
__declspec(noreturn) void overflow(void); // never return on overflow
int negate(int x)
{
    if (x == 0x80000000) overflow();
    return -x;
}
```

5.26.4 See also

- [__attribute__\(\(noreturn\)\) function attribute](#) on page 5-54.

5.27 `__declspec(nothrow)`

The `__declspec(nothrow)` attribute asserts that a call to a function never results in a C++ exception being propagated from the call into the caller.

The ARM library headers automatically add this qualifier to declarations of C functions that, according to the ISO C Standard, can never throw.

5.27.1 Usage

If the compiler knows that a function can never throw out, it might be able to generate smaller exception-handling tables for callers of that function.

5.27.2 Restrictions

If a call to a function results in a C++ exception being propagated from the call into the caller, the behavior is undefined.

This modifier is ignored when not compiling with exceptions enabled.

5.27.3 Example

```
struct S
{
    ~S();
};
__declspec(nothrow) extern void f(void);
void g(void)
{
    S s;
    f();
}
```

5.27.4 See also

- [--force_new_nothrow, --no_force_new_nothrow](#) on page 3-94
- [Using the ::operator new function](#) on page 6-15.

5.28 `__declspec(notshared)`

The `__declspec(notshared)` attribute prevents a specific class from having its virtual functions table and RTTI exported. This holds true regardless of other options you apply. For example, the use of `--export_all_vtbl` does not override `__declspec(notshared)`.

5.28.1 Example

```
struct __declspec(notshared) X
{
    virtual int f();
};
int X::f()
{
    return 1;
}
struct Y : X
{
    virtual int g();
};
int Y::g()
{
    return 1;
}
```

// do not export this

// do export this

5.29 `__declspec(thread)`

The `__declspec(thread)` attribute asserts that variables are thread-local and have *thread storage duration*, so that the linker arranges for the storage to be allocated automatically when a thread is created.

———— **Note** —————

The keyword `__thread` is supported as a synonym for `__declspec(thread)`.

5.29.1 Restrictions

File-scope thread-local variables cannot be dynamically initialized.

5.29.2 Example

```
__declspec(thread) int i;  
__thread int j;           // same as __declspec(thread) int j;
```

5.30 Function attributes

The `__attribute__` keyword enables you to specify special attributes of variables or structure fields, functions, and types. The keyword format is either of the following:

```
__attribute__((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

For example:

```
void * Function_Attributes_malloc_0(int b) __attribute__((malloc));
static int b __attribute__((__unused__));
```

Table 5-3 summarizes the available function attributes.

Table 5-3 Function attributes supported by the compiler and their equivalents

Function attribute	Non-attribute equivalent
<code>__attribute__((alias))</code>	-
<code>__attribute__((always_inline))</code>	<code>__forceinline</code>
<code>__attribute__((const))</code>	<code>__pure</code>
<code>__attribute__((constructor[[priority]]))</code>	-
<code>__attribute__((deprecated))</code>	-
<code>__attribute__((destructor[[priority]]))</code>	-
<code>__attribute__((format_arg(string-index)))</code>	-
<code>__attribute__((malloc))</code>	-
<code>__attribute__((noinline))</code>	<code>__declspec(noinline)</code>
<code>__attribute__((no_instrument_function))</code>	-
<code>__attribute__((nomerge))</code>	-
<code>__attribute__((nonnull))</code>	-
<code>__attribute__((noreturn))</code>	<code>__declspec(noreturn)</code>
<code>__attribute__((notailcall))</code>	-
<code>__attribute__((pcs("calling_convention")))</code>	-
<code>__attribute__((pure))</code>	-
<code>__attribute__((section("name")))</code>	-
<code>__attribute__((unused))</code>	-
<code>__attribute__((used))</code>	-
<code>__attribute__((visibility("visibility_type")))</code>	-
<code>__attribute__((weak))</code>	<code>__weak</code>
<code>__attribute__((weakref("target")))</code>	-

5.30.1 Usage

You can set these function attributes in the declaration, the definition, or both. For example:

```
void AddGlobals(void) __attribute__((always_inline));  
__attribute__((always_inline)) void AddGlobals(void) {...}
```

When function attributes conflict, the compiler uses the safer or stronger one. For example, `__attribute__((used))` is safer than `__attribute__((unused))`, and `__attribute__((noinline))` is safer than `__attribute__((always_inline))`.

5.31 `__attribute__((alias))` function attribute

This function attribute enables you to specify multiple aliases for functions.

Where a function is defined in the current translation unit, the alias call is replaced by a call to the function, and the alias is emitted alongside the original name. Where a function is not defined in the current translation unit, the alias call is replaced by a call to the real function. Where a function is defined as **static**, the function name is replaced by the alias name and the function is declared external if the alias name is declared external.

Note

This function attribute is a GNU compiler extension supported by the ARM compiler.

Note

Variables names might also be aliased using the corresponding variable attribute `__attribute__((alias))`.

5.31.1 Syntax

```
return-type newname([argument-list]) __attribute__((alias("oldname")));
```

Where:

oldname is the name of the function to be aliased
newname is the new name of the aliased function.

5.31.2 Example

```
#include <stdio.h>
void foo(void)
{
    printf("%s\n", __FUNCTION__);
}
void bar(void) __attribute__((alias("foo")));
void gazonk(void)
{
    bar(); // calls foo
}
```

5.31.3 See also

- [__attribute__\(\(alias\)\) variable attribute](#) on page 5-71.

5.32 `__attribute__((always_inline))` function attribute

This function attribute indicates that a function must be inlined.

The compiler attempts to inline the function, regardless of the characteristics of the function. However, the compiler does not inline a function if doing so causes problems. For example, a recursive function is inlined into itself only once.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports. It has the keyword equivalent `__forceinline`.

5.32.1 Example

```
static int max(int x, int y) __attribute__((always_inline));
static int max(int x, int y)
{
    return x > y ? x : y; // always inline if possible
}
```

5.32.2 See also

- [--forceinline](#) on page 3-95
- [__forceinline](#) on page 5-10.

5.33 `__attribute__((const))` function attribute

Many functions examine only the arguments passed to them, and have no effects except for the return value. This is a much stricter class than `__attribute__((pure))`, because a function is not permitted to read global memory. If a function is known to operate only on its arguments then it can be subject to common sub-expression elimination and loop optimizations.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports. It has the keyword equivalent `__pure`.

5.33.1 Example

```
int Function_Attributes_const_0(int b) __attribute__((const));
int Function_Attributes_const_0(int b)
{
    int aLocal=0;
    aLocal += Function_Attributes_const_0(b);
    aLocal += Function_Attributes_const_0(b);
    return aLocal;
}
```

In this code `Function_Attributes_const_0` might be called once only, with the result being doubled to obtain the correct return value.

5.33.2 See also

- [__attribute__\(\(pure\)\) function attribute on page 5-57](#)
- [Functions that return the same result when called with the same arguments on page 6-25 in *Using the Compiler*.](#)

5.34 `__attribute__((constructor[priority]))` function attribute

This attribute causes the function it is associated with to be called automatically before `main()` is entered.

———— **Note** ————

This attribute is a GNU compiler extension supported by the ARM compiler.

5.34.1 Syntax

```
__attribute__((constructor[priority]))
```

Where *priority* is an optional integer value denoting the priority. A constructor with a low integer value runs before a constructor with a high integer value. A constructor with a priority runs before a constructor without a priority.

Priority values up to and including 100 are reserved for internal use. If you use these values, the compiler gives a warning. Priority values above 100 are not reserved.

5.34.2 Usage

You can use this attribute for start-up or initialization code. For example, to specify a function that is to be called when a DLL is loaded.

This attribute can be preferable to the linker option `--init=symbol` if you are using GNU makefiles unmodified to build with the ARM compiler. That is, if you are using `--translate_gcc`, `--translate_gld`, or `--translate_g++`.

5.34.3 Example

In the following example, the constructor functions are called before execution enters `main()`, in the order specified:

```
int my_constructor(void) __attribute__((constructor));
int my_constructor2(void) __attribute__((constructor(102)));
int my_constructor3(void) __attribute__((constructor(101)));

int my_constructor(void) /* This is the 3rd constructor */
{
    /* function to be called */
    ...
    return 0;
}

int my_constructor2(void) /* This is the 1st constructor */
{
    /* function to be called */
    ...
    return 0;
}

int my_constructor3(void) /* This is the 2nd constructor */
{
    /* function to be called */
    ...
    return 0;
}
```

5.34.4 See also

- [__attribute__\(\(destructor\[*priority*\]\)\) function attribute](#) on page 5-47
- [--init=*symbol*](#) on page 2-84 in the *Linker Reference*

- `--translate_gcc` on page 3-200
- `--translate_gld` on page 3-202
- `--translate_g++` on page 3-198.

5.35 `__attribute__((deprecated))` function attribute

This function attribute indicates that a function exists but the compiler must generate a warning if the deprecated function is used.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports.

5.35.1 Example

```
int Function_Attributes_deprecated_0(int b) __attribute__((deprecated));
```

5.36 `__attribute__((destructor[priority]))` function attribute

This attribute causes the function it is associated with to be called automatically after `main()` completes or after `exit()` is called.

———— **Note** —————

This attribute is a GNU compiler extension supported by the ARM compiler.

5.36.1 Syntax

```
__attribute__((destructor[priority]))
```

Where *priority* is an optional integer value denoting the priority. A destructor with a high integer value runs before a destructor with a low value. A destructor with a priority runs before a destructor without a priority.

Priority values up to and including 100 are reserved for internal use. If you use these values, the compiler gives a warning. Priority values above 100 are not reserved.

5.36.2 Usage

This attribute can be preferable to the linker option `--fini=symbol` if you are using GNU makefiles unmodified to build with the ARM compiler. That is, if you are using `--translate_gcc`, `--translate_gld`, or `--translate_g++`.

5.36.3 Example

```
int my_destructor(void) __attribute__((destructor));

int my_destructor(void) /* This function is called after main() */
{                       /* completes or after exit() is called. */
    ...
    return 0;
}
```

5.36.4 See also

- [__attribute__\(\(constructor\[*priority*\]\)\) function attribute](#) on page 5-44
- [--fini=*symbol*](#) on page 2-70 in the *Linker Reference*
- [--translate_gcc](#) on page 3-200
- [--translate_gld](#) on page 3-202
- [--translate_g++](#) on page 3-198.

5.37 `__attribute__((format_arg(string-index)))` function attribute

This function attribute specifies that a user-defined function modifies format strings. Use of this attribute enables calls to functions like `printf()`, `scanf()`, `strftime()` or `strfmon()`, whose operands are a call to the user-defined function, to be checked for errors.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports.

5.38 `__attribute__((malloc))` function attribute

This function attribute indicates that the function can be treated like `malloc` and the compiler can perform the associated optimizations.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports.

5.38.1 Example

```
void * Function_Attributes_malloc_0(int b) __attribute__((malloc));
```

5.39 `__attribute__((noinline))` function attribute

This function attribute suppresses the inlining of a function at the call points of the function.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports. It has the `__declspec` equivalent `__declspec(noinline)`.

5.39.1 Example

```
int fn(void) __attribute__((noinline));

int fn(void)
{
    return 42;
}
```

5.39.2 See also

- [#pragma inline, #pragma no_inline](#) on page 5-101
- [__attribute__\(\(noinline\)\) constant variable attribute](#) on page 5-75
- [__declspec\(noinline\)](#) on page 5-34.

5.40 `__attribute__((no_instrument_function))` function attribute

Functions marked with this attribute are not profiled by `--gnu_instrument`.

5.40.1 See also

- [--gnu_instrument, --no_gnu_instrument](#) on page 3-109.

5.41 `__attribute__((nomerge))` function attribute

This function attribute prevents calls to the function that are distinct in the source from being combined in the object code.

5.41.1 See also

- [__attribute__\(\(notailcall\)\) function attribute](#) on page 5-55
- [--retain=option](#) on page 3-184.

5.42 `__attribute__((nonnull))` function attribute

This function attribute specifies function parameters that are not supposed to be null pointers. This enables the compiler to generate a warning on encountering such a parameter.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports.

5.42.1 Syntax

```
__attribute__((nonnull(arg-index, ...)))
```

Where *arg-index*, ... denotes the argument index list.

If no argument index list is specified, all pointer arguments are marked as nonnull.

5.42.2 Example

The following declarations are equivalent:

```
void * my_memcpy (void *dest, const void *src, size_t len) __attribute__((nonnull (1, 2)));
```

```
void * my_memcpy (void *dest, const void *src, size_t len) __attribute__((nonnull));
```

5.43 `__attribute__((noreturn))` function attribute

This function attribute informs the compiler that the function does not return. The compiler can then perform optimizations by removing the code that is never reached.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports. It has the `__declspec` equivalent `__declspec(noreturn)`. However, `__attribute__((noreturn))` and `__declspec(noreturn)` differ in that when compiling a function definition, if the function reaches an explicit or implicit return, `__attribute__((noreturn))` is ignored and the compiler generates a warning. This does not apply to `__declspec(noreturn)`.

5.43.1 Example

```
int Function_Attributes_NoReturn_0(void) __attribute__((noreturn));
```

5.43.2 See also

- [__declspec\(noreturn\)](#) on page 5-35.

5.44 `__attribute__((notailcall))` function attribute

This function attribute prevents tailcalling of the function. That is, the function is always called with a branch-and-link, even if (because the call occurs at the end of a function) the branch-and-link could be converted to a branch.

5.44.1 See also

- [__attribute__\(\(nomerge\)\)](#) function attribute on page 5-52
- [--retain=option](#) on page 3-184.

5.45 `__attribute__((pcs("calling_convention")))`

This function attribute specifies the calling convention on targets with hardware floating-point, as an alternative to the `__softfp` keyword.

———— **Note** —————

This function attribute is a GNU compiler extension supported by the ARM compiler.

5.45.1 Syntax

```
__attribute__((pcs("calling_convention")))
```

Where *calling_convention* is one of the following:

`aapcs` uses integer registers, as for `__softfp`.

`aapcs-vfp` uses floating-point registers.

5.45.2 See also

- [__softfp](#) on page 5-22
- [Compiler support for floating-point computations and linkage](#) on page 6-65 in *Using the Compiler*.

5.46 `__attribute__((pure))` function attribute

Many functions have no effects except to return a value, and their return value depends only on the parameters and global variables. Functions of this kind can be subject to data flow analysis and might be eliminated.

———— **Note** ————

This function attribute is a GNU compiler extension that the ARM compiler supports.

Although related, this function attribute is *not* equivalent to the `__pure` keyword. The function attribute equivalent to `__pure` is `__attribute__((const))`.

5.46.1 Example

```
int Function_Attributes_pure_0(int b) __attribute__((pure));
int Function_Attributes_pure_0(int b)
{
    return b++;
}

int foo(int b)
{
    int aLocal=0;
    aLocal += Function_Attributes_pure_0(b);
    aLocal += Function_Attributes_pure_0(b);
    return 0;
}
```

The call to `Function_Attributes_pure_0` in this example might be eliminated because its result is not used.

5.47 `__attribute__((section("name")))` function attribute

The section function attribute enables you to place code in different sections of the image.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports.

5.47.1 Example

In the following example, `Function_Attributes_section_0` is placed into the RO section `new_section` rather than `.text`.

```
void Function_Attributes_section_0 (void)
  __attribute__((section ("new_section")));
void Function_Attributes_section_0 (void)
{
    static int aStatic =0;
    aStatic++;
}
```

In the following example, section function attribute overrides the `#pragma arm section` setting.

```
#pragma arm section code="foo"
int f2()
{
    return 1;
} // into the 'foo' area
__attribute__((section ("bar"))) int f3()
{
    return 1;
} // into the 'bar' area
int f4()
{
    return 1;
} // into the 'foo' area
#pragma arm section
```

5.47.2 See also

- [#pragma arm section \[section_type_list\]](#) on page 5-88.

5.48 `__attribute__((unused))` function attribute

The unused function attribute prevents the compiler from generating warnings if the function is not referenced. This does not change the behavior of the unused function removal process.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports.

5.48.1 Example

```
static int Function_Attributes_unused_0(int b) __attribute__((unused));
```


5.49 `__attribute__((used))` function attribute

This function attribute informs the compiler that a static function is to be retained in the object file, even if it is unreferenced.

Static functions marked as used are emitted to a single section, in the order they are declared. You can specify the section functions are placed in using `__attribute__((section("name")))`.

Functions marked with `__attribute__((used))` are tagged in the object file to avoid removal by linker unused section removal.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports.

———— **Note** —————

Static variables can also be marked as used using `__attribute__((used))`.

5.49.1 Example

```
static int lose_this(int);
static int keep_this(int) __attribute__((used)); // retained in object file
static int keep_this_too(int) __attribute__((used)); // retained in object file
```

5.49.2 See also

- [__attribute__\(\(section\("name"\)\)\) function attribute](#) on page 5-58.
- [__attribute__\(\(used\)\) variable attribute](#) on page 5-80
- [Elimination of unused sections](#) on page 5-4 in *Using the Linker*.

5.50 `__attribute__((visibility("visibility_type")))` function attribute

This function attribute affects the visibility of ELF symbols.

———— **Note** —————

This attribute is a GNU compiler extension supported by the ARM compiler.

5.50.1 Syntax

```
__attribute__((visibility("visibility_type")))
```

Where *visibility_type* is one of the following:

default	The assumed visibility of symbols can be changed by other options. Default visibility overrides such changes. Default visibility corresponds to external linkage.
hidden	The symbol is not placed into the dynamic symbol table, so no other executable or shared library can directly reference it. Indirect references are possible using function pointers.
internal	Unless otherwise specified by the <i>processor-specific Application Binary Interface</i> (psABI), internal visibility means that the function is never called from another module.
protected	The symbol is placed into the dynamic symbol table, but references within the defining module bind to the local symbol. That is, the symbol cannot be overridden by another module.

5.50.2 Usage

Except when specifying default visibility, this attribute is intended for use with declarations that would otherwise have external linkage.

You can apply this attribute to functions and variables in C and C++. In C++, it can also be applied to class, struct, union, and enum types, and namespace declarations.

5.50.3 Example

```
void __attribute__((visibility("internal"))) foo()
{
    ...
}
```

5.50.4 See also

- [--arm_linux](#) on page 3-16
- [--visibility_inlines_hidden](#) on page 3-217
- [--hide_all, --no_hide_all](#) on page 3-113
- [__attribute__\(\(visibility\("visibility_type"\)\)\) variable attribute](#) on page 5-81.

5.51 `__attribute__((weak))` function attribute

Functions defined with `__attribute__((weak))` export their symbols weakly.

Functions declared with `__attribute__((weak))` and then defined without `__attribute__((weak))` behave as *weak* functions. This is not the same behavior as the `__weak` keyword.

———— **Note** —————

This function attribute is a GNU compiler extension that the ARM compiler supports.

5.51.1 Example

```
extern int Function_Attributes_weak_0 (int b) __attribute__((weak));
```

5.51.2 See also

- [__weak](#) on page 5-27.

5.52 `__attribute__((weakref("target")))` function attribute

This function attribute marks a function declaration as an alias that does not by itself require a function definition to be given for the target symbol.

———— **Note** —————

This function attribute is a GNU compiler extension supported by the ARM compiler.

5.52.1 Syntax

```
__attribute__((weakref("target")))
```

Where *target* is the target symbol.

5.52.2 Example

In the following example, `foo()` calls `y()` through a weak reference:

```
extern void y(void);
static void x(void) __attribute__((weakref("y")));
void foo (void)
{
    ...
    x();
    ...
}
```

5.52.3 Restrictions

This attribute can only be used on functions with internal linkage.

5.53 Type attributes

The `__attribute__` keyword enables you to specify special attributes of variables or structure fields, functions, and types. The keyword format is either of the following:

```
__attribute__((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

For example:

```
void * Function_Attributes_malloc_0(int b) __attribute__((malloc));
static int b __attribute__((__unused__));
```

Table 5-4 summarizes the available type attributes.

Table 5-4 Type attributes supported by the compiler and their equivalents

Type attribute	Non-attribute equivalent
<code>__attribute__((bitband))</code>	-
<code>__attribute__((aligned))</code>	<code>__align</code>
<code>__attribute__((packed))</code>	<code>__packed^a</code>
<code>__attribute__((transparent_union))</code>	-

a. The `__packed` qualifier does not affect type in GNU mode.

5.54 `__attribute__((bitband))` type attribute

`__attribute__((bitband))` is a type attribute that gives you efficient atomic access to single-bit values in SRAM and Peripheral regions of the memory architecture. It is possible to set or clear a single bit directly with a single memory access in certain memory regions, rather than having to use the traditional read, modify, write approach. It is also possible to read a single bit directly rather than having to use the traditional read then shift and mask operation. [Example 5-6](#) illustrates the use of `__attribute__((bitband))`.

Example 5-6 Using `__attribute__((bitband))`

```
typedef struct {
    int i : 1;
    int j : 2;
    int k : 3;
} BB __attribute__((bitband));

BB bb __attribute__((at(0x20000004)));

void foo(void)
{
    bb.i = 1;
}
```

For peripherals that are sensitive to the memory access width, byte, halfword, and word stores or loads to the alias space are generated for `char`, `short`, and `int` types of bitfields of bit-banded structs respectively.

In [Example 5-7](#), bit-banded access is generated for `bb.i`.

Example 5-7 Bitfield bit-band access

```
typedef struct {
    char i : 1;
    int j : 2;
    int k : 3;
} BB __attribute__((bitband));

BB bb __attribute__((at(0x20000004)));

void foo()
{
    bb.i = 1;
}
```

If you do not use `__attribute__((at()))` to place the bit-banded variable in the bit-band region, you must relocate it using another method. You can do this by either using an appropriate scatter-loading description file or by using the `--rw_base` linker command-line option. See the *Linker Reference* for more information.

5.54.1 Restrictions

The following restrictions apply:

- This type attribute can only be used with `struct`. Any union type or other aggregate type with a union as a member cannot be bit-banded.

- Members of structs cannot be bit-banded individually.
- Bit-banded accesses are only generated for single-bit bitfields.
- Bit-banded accesses are not generated for **const** objects, pointers, and local objects.
- Bit-banding is only available on some processors. For example, the Cortex-M3 and Cortex-M4 processors.

5.54.2 See also

- [__attribute__\(\(at\(address\)\)\) variable attribute](#) on page 5-72
- [--bitband](#) on page 3-28
- the *Technical Reference Manual* for your processor.

5.55 `__attribute__((aligned))` type attribute

The `aligned` type attribute specifies a minimum alignment for the type.

———— **Note** —————

This type attribute is a GNU compiler extension that the ARM compiler supports.

5.56 `__attribute__((packed))` type attribute

The packed type attribute specifies that a type must have the smallest possible alignment.

———— **Note** —————

This type attribute is a GNU compiler extension that the ARM compiler supports.

5.56.1 Errors

Taking the address of a field with the packed attribute or in a structure with the packed attribute yields a `__packed`-qualified pointer. The compiler will produce a type error if you attempt to implicitly cast this pointer to a non-`__packed` pointer. This contrasts with its behavior for address-taken fields of a `#pragma packed` structure.

The compiler generates a warning message if you use this attribute in a typedef.

5.56.2 See also

- [__packed](#) on page 5-17
- [__attribute__\(\(packed\)\) variable attribute](#) on page 5-76
- [#pragma pack\(n\)](#) on page 5-107
- [Packed structures](#) on page 6-11
- [The __packed qualifier and unaligned data access in C and C++ code](#) on page 6-47 in *Using the Compiler*
- [Comparisons of an unpacked struct, a __packed struct, and a struct with individually __packed fields, and of a __packed struct and a #pragma packed struct](#) on page 6-52 in *Using the Compiler*.

5.57 `__attribute__((transparent_union))` type attribute

The `transparent_union` type attribute enables you to specify a *transparent union type*, that is, a union data type qualified with `__attribute__((transparent_union))`.

When a function is defined with a parameter having transparent union type, a call to the function with an argument of any type in the union results in the initialization of a union object whose member has the type of the passed argument and whose value is set to the value of the passed argument.

When a union data type is qualified with `__attribute__((transparent_union))`, the transparent union applies to all function parameters with that type.

———— Note —————

This type attribute is a GNU compiler extension that the ARM compiler supports.

———— Note —————

Individual function parameters might also be qualified with the corresponding `__attribute__((transparent_union))` variable attribute.

5.57.1 Example

```
typedef union { int i; float f; } U __attribute__((transparent_union));
void foo(U u)
{
    static int s;
    s += u.i;    /* Use the 'int' field */
}
void caller(void)
{
    foo(1);      /* u.i is set to 1 */
    foo(1.0f);   /* u.f is set to 1.0f */
}
```

5.57.2 Mode

Supported in GNU mode only.

5.57.3 See also

- [__attribute__\(\(transparent_union\)\) variable attribute on page 5-78.](#)

5.58 Variable attributes

The `__attribute__` keyword enables you to specify special attributes of variables or structure fields, functions, and types. The keyword format is either of the following:

```
__attribute__((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

For example:

```
void * Function_Attributes_malloc_0(int b) __attribute__((malloc));
static int b __attribute__((__unused__));
```

Table 5-5 summarizes the available variable attributes.

Table 5-5 Variable attributes supported by the compiler and their equivalents

Variable attribute	Non-attribute equivalent
<code>__attribute__((alias))</code>	-
<code>__attribute__((at(address)))</code>	-
<code>__attribute__((aligned))</code>	-
<code>__attribute__((deprecated))</code>	-
<code>__attribute__((noinline))</code>	-
<code>__attribute__((packed))</code>	-
<code>__attribute__((section("name")))</code>	-
<code>__attribute__((transparent_union))</code>	-
<code>__attribute__((unused))</code>	-
<code>__attribute__((used))</code>	-
<code>__attribute__((visibility("visibility_type")))</code>	-
<code>__attribute__((weak))</code>	<code>__weak</code>
<code>__attribute__((weakref("target")))</code>	-
<code>__attribute__((zeroinit))</code>	-

5.59 `__attribute__((alias))` variable attribute

This variable attribute enables you to specify multiple aliases for variables.

Where a variable is defined in the current translation unit, the alias reference is replaced by a reference to the variable, and the alias is emitted alongside the original name. Where a variable is not defined in the current translation unit, the alias reference is replaced by a reference to the real variable. Where a variable is defined as **static**, the variable name is replaced by the alias name and the variable is declared external if the alias is declared external.

———— **Note** —————

Function names might also be aliased using the corresponding function attribute `__attribute__((alias))`.

5.59.1 Syntax

```
type newname __attribute__((alias("oldname")));
```

Where:

oldname is the name of the variable to be aliased
newname is the new name of the aliased variable.

5.59.2 Example

```
#include <stdio.h>
int oldname = 1;
extern int newname __attribute__((alias("oldname"))); // declaration
void foo(void)
{
    printf("newname = %d\n", newname); // prints 1
}
```

5.59.3 See also

- [__attribute__\(\(alias\)\) function attribute](#) on page 5-41.

5.60 `__attribute__((at(address)))` variable attribute

This variable attribute enables you to specify the absolute address of a variable.

The variable is placed in its own section, and the section containing the variable is given an appropriate type by the compiler:

- Read-only variables are placed in a section of type RO.
- Initialized read-write variables are placed in a section of type RW.
Variables explicitly initialized to zero are placed in:
 - A section of type ZI in RVCT 4.0 and later.
 - A section of type RW (not ZI) in RVCT 3.1 and earlier. Such variables are not candidates for the ZI-to-RW optimization of the compiler.
- Uninitialized variables are placed in a section of type ZI.

———— **Note** —————

This variable attribute is not supported by GNU compilers.

5.60.1 Syntax

```
__attribute__((at(address)))
```

Where:

address is the desired address of the variable.

5.60.2 Restrictions

The linker is not always able to place sections produced by the `at` variable attribute.

The compiler faults use of the `at` attribute when it is used on declarations with incomplete types.

5.60.3 Errors

The linker gives an error message if it is not possible to place a section at a specified address.

5.60.4 Examples

```
const int x1 __attribute__((at(0x10000))) = 10; /* RO */
int x2 __attribute__((at(0x12000))) = 10; /* RW */
int x3 __attribute__((at(0x14000))) = 0; /* RVCT 3.1 and earlier: RW.
                                     * RVCT 4.0 and later: ZI. */
int x4 __attribute__((at(0x16000))); /* ZI */
```

5.60.5 See also

- [Using `__at` sections to place sections at a specific address on page 8-37](#) in *Using the Linker*.

5.61 `__attribute__((aligned))` variable attribute

The aligned variable attribute specifies a minimum alignment for the variable or structure field, measured in bytes.

———— **Note** —————

This variable attribute is a GNU compiler extension that the ARM compiler supports.

5.61.1 Examples

```
/* Aligns on 16-byte boundary */  
int x __attribute__((aligned (16)));
```

```
/* In this case, the alignment used is the maximum alignment for a scalar data type.  
For ARM, this is 8 bytes. */  
short my_array[3] __attribute__((aligned));
```

5.61.2 See also

- [__align](#) on page 5-6.

5.62 `__attribute__((deprecated))` variable attribute

The deprecated variable attribute enables the declaration of a deprecated variable without any warnings or errors being issued by the compiler. However, any access to a deprecated variable creates a warning but still compiles. The warning gives the location where the variable is used and the location where it is defined. This helps you to determine why a particular definition is deprecated.

———— **Note** —————

This variable attribute is a GNU compiler extension that the ARM compiler supports.

5.62.1 Example

```
extern int Variable_Attributes_deprecated_0 __attribute__((deprecated));
extern int Variable_Attributes_deprecated_1 __attribute__((deprecated));
void Variable_Attributes_deprecated_2()
{
    Variable_Attributes_deprecated_0=1;
    Variable_Attributes_deprecated_1=2;
}
```

Compiling this example generates two warning messages.

5.63 `__attribute__((noinline))` constant variable attribute

The `noinline` variable attribute prevents the compiler from making any use of a constant data value for optimization purposes, without affecting its placement in the object. This feature can be used for patchable constants, that is, data that is later patched to a different value. It is an error to try to use such constants in a context where a constant value is required. For example, an array dimension.

5.63.1 Example

```
__attribute__((noinline)) const int m = 1;
```

5.63.2 See also

- [#pragma inline, #pragma no_inline](#) on page 5-101
- [__attribute__\(\(noinline\)\) function attribute](#) on page 5-50
- [__declspec\(noinline\)](#) on page 5-34.

5.64 `__attribute__((packed))` variable attribute

The packed variable attribute specifies that a variable or structure field has the smallest possible alignment. That is, one byte for a variable, and one bit for a field, unless you specify a larger value with the `aligned` attribute.

———— **Note** —————

This variable attribute is a GNU compiler extension that the ARM compiler supports.

5.64.1 Example

```
struct
{
    char a;
    int b __attribute__((packed));
} Variable_Attributes_packed_0;
```

5.64.2 See also

- [__packed](#) on page 5-17
- [__attribute__\(\(packed\)\)](#) type attribute on page 5-68
- [#pragma pack\(n\)](#) on page 5-107
- [Packed structures](#) on page 6-11
- [The __packed qualifier and unaligned data access in C and C++ code](#) on page 6-47 in *Using the Compiler*
- [Comparisons of an unpacked struct, a __packed struct, and a struct with individually __packed fields, and of a __packed struct and a #pragma packed struct](#) on page 6-52 in *Using the Compiler*.

5.65 `__attribute__((section("name")))` variable attribute

Normally, the ARM compiler places the objects it generates in sections like `.data` and `.bss`. However, you might require additional data sections or you might want a variable to appear in a special section, for example, to map to special hardware. The section attribute specifies that a variable must be placed in a particular data section. If you use the section attribute, read-only variables are placed in RO data sections, read-write variables are placed in RW data sections unless you use the `zero_init` attribute. In this case, the variable is placed in a ZI section.

———— Note ————

This variable attribute is a GNU compiler extension supported by the ARM compiler.

5.65.1 Example

```
/* in RO section */
const int descriptor[3] __attribute__((section ("descr"))) = { 1,2,3 };

/* in RW section */
long long rw_initialized[10] __attribute__((section ("INITIALIZED_RW"))) = {5};

/* in RW section */
long long rw[10] __attribute__((section ("RW")));

/* in ZI section */
long long altstack[10] __attribute__((section ("STACK"), zero_init));
```

5.65.2 See also

- [How to find where a symbol is placed when linking on page 6-6 in Using the Linker](#)
- [Using fromelf to find where a symbol is placed in an executable ELF image on page 3-12 in Using the fromelf Image Converter.](#)

5.66 `__attribute__((transparent_union))` variable attribute

The `transparent_union` variable attribute, attached to a function parameter that is a union, means that the corresponding argument can have the type of any union member, but the argument is passed as if its type were that of the first union member.

Note

The C specification states that the value returned when a union is written as one type and read back with another is undefined. Therefore, a method of distinguishing which type a `transparent_union` is written in must also be passed as an argument.

Note

This variable attribute is a GNU compiler extension that the ARM compiler supports.

Note

You can also use this attribute on a typedef for a union data type. In this case it applies to all function parameters with that type.

5.66.1 Mode

Supported in GNU mode only.

5.66.2 Example

```
typedef union
{
    int myint;
    float myfloat;
} transparent_union_t;
void Variable_Attributes_transparent_union_0(transparent_union_t aUnion
__attribute__((transparent_union)))
{
    static int aStatic;
    aStatic +=aUnion.myint;
}
void Variable_Attributes_transparent_union_1()
{
    int aLocal =0;
    float bLocal =0;
    Variable_Attributes_transparent_union_0(aLocal);
    Variable_Attributes_transparent_union_0(bLocal);
}
```

5.66.3 See also

- [__attribute__\(\(transparent_union\)\) type attribute](#) on page 5-69.

5.67 `__attribute__((unused))` variable attribute

Normally, the compiler warns if a variable is declared but is never referenced. This attribute informs the compiler that you expect a variable to be unused and tells it not issue a warning if it is not used.

———— **Note** —————

This variable attribute is a GNU compiler extension that the ARM compiler supports.

5.67.1 Example

```
void Variable_Attributes_unused_0()
{
    static int aStatic =0;
    int aUnused __attribute__((unused));
    int bUnused;
    aStatic++;
}
```

In this example, the compiler warns that `bUnused` is declared but never referenced, but does not warn about `aUnused`.

———— **Note** —————

The GNU compiler does not give any warning.

5.68 `__attribute__((used))` variable attribute

This variable attribute informs the compiler that a static variable is to be retained in the object file, even if it is unreferenced.

Static variables marked as used are emitted to a single section, in the order they are declared. You can specify the section that variables are placed in using `__attribute__((section("name")))`.

Data marked with `__attribute__((used))` is tagged in the object file to avoid removal by linker unused section removal.

———— **Note** —————

This variable attribute is a GNU compiler extension that the ARM compiler supports.

———— **Note** —————

Static functions can also be marked as used using `__attribute__((used))`.

5.68.1 Usage

You can use `__attribute__((used))` to build tables in the object.

5.68.2 Example

```
static int lose_this = 1;
static int keep_this __attribute__((used)) = 2;    // retained in object file
static int keep_this_too __attribute__((used)) = 3; // retained in object file
```

5.68.3 See also

- [__attribute__\(\(section\("name"\)\)\) variable attribute](#) on page 5-77
- [__attribute__\(\(used\)\) function attribute](#) on page 5-60
- [Elimination of unused sections](#) on page 5-4 in *Using the Linker*.

5.69 `__attribute__((visibility("visibility_type")))` variable attribute

This variable attribute affects the visibility of ELF symbols.

———— Note ————

This attribute is a GNU compiler extension supported by the ARM compiler.

5.69.1 Syntax

```
__attribute__((visibility("visibility_type")))
```

Where *visibility_type* is one of the following:

default	The assumed visibility of symbols can be changed by other options. Default visibility overrides such changes. Default visibility corresponds to external linkage.
hidden	The symbol is not placed into the dynamic symbol table, so no other executable or shared library can directly reference it. Indirect references are possible using function pointers.
internal	Unless otherwise specified by the <i>processor-specific Application Binary Interface</i> (psABI), internal visibility means that the function is never called from another module.
protected	The symbol is placed into the dynamic symbol table, but references within the defining module bind to the local symbol. That is, the symbol cannot be overridden by another module.

5.69.2 Usage

Except when specifying default visibility, this attribute is intended for use with declarations that would otherwise have external linkage.

You can apply this attribute to functions and variables in C and C++. In C++, you can also apply it to `class`, `struct`, `union`, and `enum` types, and `namespace` declarations.

5.69.3 Example

```
int i __attribute__((visibility("hidden")));
```

5.69.4 See also

- [--arm_linux](#) on page 3-16
- [--hide_all, --no_hide_all](#) on page 3-113
- [__attribute__\(\(visibility\("visibility_type"\)\)\) function attribute](#) on page 5-61.

5.70 `__attribute__((weak))` variable attribute

The declaration of a weak variable is permitted, and acts in a similar way to `__weak`.

- in GNU mode:

```
extern int Variable_Attributes_weak_1 __attribute__((weak));
```
- the equivalent in non-GNU mode is:

```
__weak int Variable_Attributes_weak_compare;
```

Note

The `extern` qualifier is required in GNU mode. In non-GNU mode the compiler assumes that if the variable is not `extern` then it is treated like any other non weak variable.

Note

This variable attribute is a GNU compiler extension that the ARM compiler supports.

5.70.1 See also

- [__weak](#) on page 5-27.

5.71 `__attribute__((weakref("target")))` variable attribute

This variable attribute marks a variable declaration as an alias that does not by itself require a definition to be given for the target symbol.

———— **Note** ————

This variable attribute is a GNU compiler extension supported by the ARM compiler.

5.71.1 Syntax

```
__attribute__((weakref("target")))
```

Where *target* is the target symbol.

5.71.2 Example

In the following example, *a* is assigned the value of *y* through a weak reference:

```
extern int y;
static int x __attribute__((weakref("y")));

void foo (void)
{
    int a = x;
    ...
}
```

5.71.3 Restrictions

This attribute can only be used on variables that are declared as static.

5.72 `__attribute__((zero_init))` variable attribute

The section attribute specifies that a variable must be placed in a particular data section. The `zero_init` attribute specifies that a variable with no initializer is placed in a ZI data section. If an initializer is specified, an error is reported.

5.72.1 Example

```
__attribute__((zero_init)) int x;           /* in section ".bss" */  
__attribute__((section("mybss"), zero_init)) int y; /* in section "mybss" */
```

5.72.2 See also

- [__attribute__\(\(section\("name"\)\)\) variable attribute on page 5-77.](#)

5.73 Pragas

The ARM compiler recognizes a number of ARM-specific pragmas.

———— **Note** ————

Pragmas override related command-line options. For example, `#pragma arm` overrides the command-line option `--thumb`.

Table 5-6 summarizes the available pragmas.

Table 5-6 Pragas supported by the compiler

Pragas		
<code>#pragma anon_unions,</code> <code>#pragma no_anon_unions</code>	<code>#pragma hdrstop</code>	<code>#pragma pack(n)</code>
<code>#pragma arm</code>	<code>#pragma import symbol_name</code>	<code>#pragma pop</code>
<code>#pragma arm section</code> <code>[section_type_list]</code>	<code>#pragma</code> <code>import(__use_full_stdio)</code>	<code>#pragma push</code>
<code>#pragma diag_default</code> <code>tag[, tag, ...]</code>	<code>#pragma</code> <code>import(__use_smaller_memcpy)</code>	<code>#pragma softfp_linkage,</code> <code>no_softfp_linkage</code>
<code>#pragma diag_error</code> <code>tag[, tag, ...]</code>	<code>#pragma inline,</code> <code>#pragma no_inline</code>	<code>#pragma unroll [(n)]</code>
<code>#pragma diag_remark</code> <code>tag[, tag, ...]</code>	<code>#pragma no_pch</code>	<code>#pragma unroll_completely</code>
<code>#pragma diag_suppress</code> <code>tag[, tag, ...]</code>	<code>#pragma Onum</code>	<code>#pragma thumb</code>
<code>#pragma diag_warning</code> <code>tag[, tag, ...]</code>	<code>#pragma once</code>	<code>#pragma weak symbol</code>
<code>#pragma</code> <code>[no_]exceptions_unwind</code>	<code>#pragma Ospace</code>	<code>#pragma weak symbol1 =</code> <code>symbol2</code>
<code>#pragma GCC system_header</code>	<code>#pragma Otime</code>	

5.74 #pragma anon_unions, #pragma no_anon_unions

These pragmas enable and disable support for anonymous structures and unions.

5.74.1 Default

The default is #pragma no_anon_unions.

5.74.2 See also

- [Anonymous classes, structures and unions](#) on page 4-38
- [__attribute__\(\(transparent_union\)\) type attribute](#) on page 5-69.

5.75 #pragma arm

This pragma switches code generation to the ARM instruction set. It overrides the `--thumb` compiler option.

5.75.1 Usage

Use `#pragma push` and `#pragma pop` on `#pragma arm` or `#pragma thumb` outside of functions, but not inside of them, to change state. This is because `#pragma arm` and `#pragma thumb` only apply at the function level. Instead, put them around the function definition.

5.75.2 See also

- [--arm](#) on page 3-15
- [--thumb](#) on page 3-197
- [#pragma thumb](#) on page 5-112.

5.76 #pragma arm section [*section_type_list*]

This pragma specifies a section name to be used for subsequent functions or objects. This includes definitions of anonymous objects the compiler creates for initializations.

———— **Note** —————

You can use `__attribute__((section(.)))` for functions or variables as an alternative to #pragma arm section.

5.76.1 Syntax

```
#pragma arm section [section_type_list]
```

Where:

section_type_list specifies an optional list of section names to be used for subsequent functions or objects. The syntax of *section_type_list* is:

```
section_type[(="name")] [, section_type="name"]*
```

Valid section types are:

- code
- rodata
- rwdata
- zidata.

5.76.2 Usage

Use #pragma arm section [*section_type_list*] to place functions and variables in separate named sections. The scatter-loading description file can then be used to locate these at a particular address in memory.

5.76.3 Restrictions

This option has no effect on:

- Inline functions and their local static variables.
- Template instantiations and their local static variables.
- Elimination of unused variables and functions. However, using #pragma arm section might enable the linker to eliminate a function or variable that might otherwise be kept because it is in the same section as a used function or variable.
- The order that definitions are written to the object file.

5.76.4 Example

```
int x1 = 5;                // in .data (default)
int y1[100];              // in .bss (default)
int const z1[3] = {1,2,3}; // in .constdata (default)
#pragma arm section rwdata = "foo", rodata = "bar"
int x2 = 5;                // in foo (data part of region)
int y2[100];              // in .bss
int const z2[3] = {1,2,3}; // in bar
char *s2 = "abc";         // s2 in foo, "abc" in .conststring
#pragma arm section rodata
int x3 = 5;                // in foo
```

```
int y3[100];                // in .bss
int const z3[3] = {1,2,3}; // in .constdata
char *s3 = "abc";          // s3 in foo, "abc" in .conststring
#pragma arm section code = "foo"
int add1(int x)             // in foo (code part of region)
{
    return x+1;
}
#pragma arm section code
```

5.76.5 See also

- [__attribute__\(\(section\("name"\)\)\) function attribute](#) on page 5-58
- [Chapter 8 Using scatter files](#) in *Using the Linker*.

5.77 #pragma diag_default tag[,tag,...]

This pragma returns the severity of the diagnostic messages that have the specified tags to the severities that were in effect before any pragmas were issued. Diagnostic messages are messages whose message numbers are postfixed by -D, for example, #550-D.

5.77.1 Syntax

```
#pragma diag_default tag[,tag,...]
```

Where:

`tag[,tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.

At least one diagnostic message number must be specified.

5.77.2 Example

```
// <stdio.h> not #included deliberately
#pragma diag_error 223
void hello(void)
{
    printf("Hello ");
}
#pragma diag_default 223
void world(void)
{
    printf("world!\n");
}
```

Compiling this code with the option `--diag_warning=223` generates diagnostic messages to report that the function `printf()` is declared implicitly.

The effect of `#pragma diag_default 223` is to return the severity of diagnostic message 223 to Warning severity, as specified by the `--diag_warning` command-line option.

5.77.3 See also

- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [#pragma diag_error tag\[,tag,...\]](#) on page 5-91
- [#pragma diag_remark tag\[,tag,...\]](#) on page 5-92
- [#pragma diag_suppress tag\[,tag,...\]](#) on page 5-93
- [#pragma diag_warning tag\[, tag, ...\]](#) on page 5-94
- [About compiler diagnostic messages on page 7-2](#) in *Using the Compiler*.

5.78 #pragma diag_error tag[,tag,...]

This pragma sets the diagnostic messages that have the specified tags to Error severity. Diagnostic messages are messages whose message numbers are postfixed by -D, for example, #550-D.

5.78.1 Syntax

```
#pragma diag_error tag[,tag,...]
```

Where:

`tag[, tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.

At least one diagnostic message number must be specified.

5.78.2 See also

- [--diag_error=tag\[,tag,...\]](#) on page 3-70
- [#pragma diag_default tag\[,tag,...\]](#) on page 5-90
- [#pragma diag_remark tag\[,tag,...\]](#) on page 5-92
- [#pragma diag_suppress tag\[,tag,...\]](#) on page 5-93
- [#pragma diag_warning tag\[, tag, ...\]](#) on page 5-94
- [Options that change the severity of compiler diagnostic messages on page 7-4](#) in *Using the Compiler*.

5.79 #pragma diag_remark tag[,tag,...]

This pragma sets the diagnostic messages that have the specified tags to Remark severity. Diagnostic messages are messages whose message numbers are postfixed by -D, for example, #550-D.

#pragma diag_remark behaves analogously to #pragma diag_errors, except that the compiler sets the diagnostic messages having the specified tags to Remark severity rather than Error severity.

Note

Remarks are not displayed by default. Use the --remarks compiler option to see remark messages.

5.79.1 Syntax

```
#pragma diag_remark tag[,tag,...]
```

Where:

`tag[,tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.

5.79.2 See also

- [--diag_remark=tag\[,tag,...\]](#) on page 3-71
- [--remarks](#) on page 3-181
- [#pragma diag_default tag\[,tag,...\]](#) on page 5-90
- [#pragma diag_error tag\[,tag,...\]](#) on page 5-91
- [#pragma diag_suppress tag\[,tag,...\]](#) on page 5-93
- [#pragma diag_warning tag\[, tag, ...\]](#) on page 5-94
- [Options that change the severity of compiler diagnostic messages on page 7-4](#) in *Using the Compiler*.

5.80 #pragma diag_suppress tag[,tag,...]

This pragma disables all diagnostic messages that have the specified tags. Diagnostic messages are messages whose message numbers are postfixed by -D, for example, #550-D.

#pragma diag_suppress behaves analogously to #pragma diag_errors, except that the compiler suppresses the diagnostic messages having the specified tags rather than setting them to have Error severity.

5.80.1 Syntax

```
#pragma diag_suppress tag[,tag,...]
```

Where:

`tag[,tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages to be suppressed.

5.80.2 See also

- [--diag_suppress=tag\[,tag,...\]](#) on page 3-73
- [#pragma diag_default tag\[,tag,...\]](#) on page 5-90
- [#pragma diag_error tag\[,tag,...\]](#) on page 5-91
- [#pragma diag_remark tag\[,tag,...\]](#) on page 5-92
- [#pragma diag_warning tag\[, tag, ...\]](#) on page 5-94
- [Chapter 7 Compiler Diagnostic Messages](#) in *Using the Compiler*.

5.81 #pragma diag_warning tag[, tag, ...]

This pragma sets the diagnostic messages that have the specified tags to Warning severity. Diagnostic messages are messages whose message numbers are postfixed by -D, for example, #550-D.

#pragma diag_warning behaves analogously to #pragma diag_errors, except that the compiler sets the diagnostic messages having the specified tags to Warning severity rather than Error severity.

5.81.1 Syntax

```
#pragma diag_warning tag[,tag,...]
```

Where:

`tag[,tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.

5.81.2 See also

- [--diag_warning=tag\[,tag,...\]](#) on page 3-75
- [#pragma diag_default tag\[,tag,...\]](#) on page 5-90
- [#pragma diag_error tag\[,tag,...\]](#) on page 5-91
- [#pragma diag_remark tag\[,tag,...\]](#) on page 5-92
- [#pragma diag_suppress tag\[,tag,...\]](#) on page 5-93
- [Options that change the severity of compiler diagnostic messages on page 7-4](#) in *Using the Compiler*.

5.82 #pragma exceptions_unwind, #pragma no_exceptions_unwind

These pragmas enable and disable function unwinding at runtime.

5.82.1 Default

The default is #pragma exceptions_unwind.

5.82.2 See also

- [--exceptions, --no_exceptions](#) on page 3-87
- [--exceptions_unwind, --no_exceptions_unwind](#) on page 3-88
- [Function unwinding at runtime](#) on page 6-22.

5.83 #pragma GCC system_header

This pragma is available in GNU mode. It causes subsequent declarations in the current file to be marked as if they occur in a system header file.

This pragma can affect the severity of some diagnostic messages.

5.83.1 See also

- [--gnu on page 3-107](#).

5.84 #pragma hdrstop

This pragma enables you to specify where the set of precompilation header files end.

This pragma must appear before the first token that does not belong to a preprocessing directive.

5.84.1 See also

- [PreCompiled Header \(PCH\) files on page 5-33](#) in *Using the Compiler*.

5.85 #pragma import *symbol_name*

This pragma generates an importing reference to *symbol_name*. This is the same as the assembler directive:

```
IMPORT symbol_name
```

5.85.1 Syntax

```
#pragma import symbol_name
```

Where:

symbol_name is a symbol to be imported.

5.85.2 Usage

You can use this pragma to select certain features of the C library, such as the heap implementation or real-time division. If a feature described in this book requires a symbol reference to be imported, the required symbol is specified.

5.85.3 See also

- [Using the C library with an application on page 2-33](#) in *Using ARM C and C++ Libraries and Floating-Point Support*.

5.86 #pragma import(__use_full_stdio)

This pragma selects an extended version of microlib that uses full standard ANSI C input and output functionality.

———— **Note** ————

Microlib is an alternative library to the default C library. Only use this pragma if you are using microlib.

The following exceptions apply:

- feof() and ferror() always return 0
- setvbuf() and setbuf() are guaranteed to fail.

feof() and ferror() always return 0 because the error and end-of-file indicators are not supported.

setvbuf() and setbuf() are guaranteed to fail because all streams are unbuffered.

This version of microlib stdio can be retargeted in the same way as the standardlib stdio functions.

5.86.1 See also

- [--library_type=lib](#) on page 3-130
- [About microlib on page 3-2](#) in *Using ARM C and C++ Libraries and Floating-Point Support*
- [Tailoring input/output functions in the C and C++ libraries on page 2-92](#) in *Using ARM C and C++ Libraries and Floating-Point Support*.

5.87 #pragma import(__use_smaller_memcpy)

This pragma selects a smaller, but slower, version of `memcpy()` for use with the C micro-library (microlib). A byte-by-byte implementation of `memcpy()` using LDRB and STRB is used.

———— **Note** —————

Microlib is an alternative library to the default C library. Only use this pragma if you are using microlib.

5.87.1 Default

The default version of `memcpy()` used by microlib is a larger, but faster, word-by-word implementation using LDR and STR.

5.87.2 See also

- [--library_type=lib](#) on page 3-130
- [Chapter 3 The ARM C micro-library](#) in *Using ARM C and C++ Libraries and Floating-Point Support*.

5.88 #pragma inline, #pragma no_inline

These pragmas control inlining, similar to the `--inline` and `--no_inline` command-line options. A function defined under `#pragma no_inline` is not inlined into other functions, and does not have its own calls inlined.

The effect of suppressing inlining into other functions can also be achieved by marking the function as `__declspec(noinline)` or `__attribute__((noinline))`.

5.88.1 Default

The default is `#pragma inline`.

5.88.2 See also

- [--inline, --no_inline](#) on page 3-122
- [__declspec\(noinline\)](#) on page 5-34
- [__attribute__\(\(noinline\)\) constant variable attribute](#) on page 5-75
- [__attribute__\(\(noinline\)\) function attribute](#) on page 5-50.

5.89 #pragma no_pch

This pragma suppresses PCH processing for a given source file.

5.89.1 See also

- [--pch](#) on page 3-165
- [PreCompiled Header \(PCH\) files](#) on page 5-33 in *Using the Compiler*.

5.90 #pragma Onum

This pragma changes the optimization level.

5.90.1 Syntax

```
#pragma Onum
```

Where:

num is the new optimization level.
The value of *num* is 0, 1, 2 or 3.

5.90.2 Usage

This pragma enables you to assign optimization levels to individual functions.

5.90.3 Restriction

The pragma must be placed outside the function.

5.90.4 See also

- [-Onum on page 3-156](#)
- [#pragma Ospace on page 5-105](#)
- [#pragma Otime on page 5-106](#).

5.91 #pragma once

This pragma enables the compiler to skip subsequent includes of that header file.

#pragma once is accepted for compatibility with other compilers, and enables you to use other forms of header guard coding. However, it is preferable to use #ifndef and #define coding because this is more portable.

5.91.1 Example

The following example shows the placement of a #ifndef guard around the body of the file, with a #define of the guard variable after the #ifndef.

```
#ifndef FILE_H
#define FILE_H
#pragma once      // optional ... body of the header file ...#endif
```

The #pragma once is marked as optional in this example. This is because the compiler recognizes the #ifndef header guard coding and skips subsequent includes even if #pragma once is absent.

5.92 #pragma Ospace

This pragma instructs the compiler to perform optimizations to reduce image size at the expense of a possible increase in execution time.

5.92.1 Usage

This pragma enables you to assign optimization goals to individual functions.

5.92.2 Restriction

The pragma must be placed outside the function.

5.92.3 See also

- [-Ospace on page 3-160](#)
- [#pragma Onum on page 5-103](#)
- [#pragma Otime on page 5-106.](#)

5.93 #pragma Otime

This pragma instructs the compiler to perform optimizations to reduce execution time at the expense of a possible increase in image size.

5.93.1 Usage

This pragma enables you to assign optimization goals to individual functions.

5.93.2 Restriction

The pragma must be placed outside the function.

5.93.3 See also

- [-Otime on page 3-161](#)
- [#pragma Onum on page 5-103](#)
- [#pragma Ospace on page 5-105.](#)

5.94 #pragma pack(*n*)

This pragma aligns members of a structure to the minimum of *n* and their natural alignment. Packed objects are read and written using unaligned accesses.

———— **Note** ————

This pragma is a GNU compiler extension that the ARM compiler supports.

5.94.1 Syntax

```
#pragma pack(n)
```

Where:

n is the alignment in bytes, valid alignment values being 1, 2, 4 and 8.

5.94.2 Default

The default is #pragma pack(8).

5.94.3 Errors

Taking the address of a field in a #pragma packed **struct** does not yield a __packed pointer, so the compiler will not produce an error if you assign this address to a non-__packed pointer. However, the field might not be properly aligned for its type, and dereferencing such an unaligned pointer results in Undefined behavior.

5.94.4 Example

This example demonstrates how pack(2) aligns integer variable b to a 2-byte boundary.

```
typedef struct
{
    char a;
    int b;
} S;

#pragma pack(2)

typedef struct
{
    char a;
    int b;
} SP;

S var = { 0x11, 0x44444444 };
SP pvar = { 0x11, 0x44444444 };
```

The layout of S is as shown in [Figure 5-1](#), while the layout of SP is as shown in [Figure 5-2 on page 5-108](#). In [Figure 5-2 on page 5-108](#), x denotes one byte of padding.

0	1	2	3
a	padding		
4	5	6	7
b	b	b	b

Figure 5-1 Nonpacked structure S

0	1	2	3
a	x	b	b
4	5		
b	b		

Figure 5-2 Packed structure SP

Note

SP is a 6-byte structure. There is no padding after b.

5.94.5 See also

- [__packed](#) on page 5-17
- [__attribute\(\(packed\)\)](#) type attribute on page 5-68
- [__attribute__\(\(packed\)\)](#) variable attribute on page 5-76
- [Packed structures](#) on page 6-11
- [The __packed qualifier and unaligned data access in C and C++ code](#) on page 6-47 in *Using the Compiler*
- [Comparisons of an unpacked struct, a __packed struct, and a struct with individually __packed fields, and of a __packed struct and a #pragma packed struct](#) on page 6-52 in *Using the Compiler*.

5.95 #pragma pop

This pragma restores the previously saved pragma state.

5.95.1 See also

- [#pragma push on page 5-110](#).

5.96 #pragma push

This pragma saves the current pragma state.

5.96.1 See also

- [#pragma pop](#) on page 5-109.

5.97 #pragma softfp_linkage, #pragma no_softfp_linkage

These pragmas control software floating-point linkage.

#pragma softfp_linkage asserts that all function declarations up to the next #pragma no_softfp_linkage describe functions that use software floating-point linkage.

———— **Note** —————

These pragmas have the keyword equivalent `__softfp`.

5.97.1 Usage

These pragmas can be useful when applied to an entire interface specification, located in the header file, without altering that file.

5.97.2 Default

The default is #pragma no_softfp_linkage.

5.97.3 See also

- [__softfp on page 5-22](#)
- [Compiler support for floating-point computations and linkage on page 6-65 in Using the Compiler.](#)

5.98 #pragma thumb

This pragma switches code generation to the Thumb instruction set. It overrides the `--arm` compiler option.

If you are compiling code for a Thumb processor without Thumb-2 technology and using VFP, *any* function containing floating-point operations is compiled for ARM.

5.98.1 Usage

Use `#pragma push` and `#pragma pop` on `#pragma arm` or `#pragma thumb` outside of functions, but not inside of them, to change state. This is because `#pragma arm` and `#pragma thumb` only apply at the function level. Instead, put them around the function definition.

Example 5-8 Use of #pragma thumb with a function

```

/* foo.c */

#pragma push      // in arm state, save current pragma state
#pragma thumb    // change to thumb state

void bar(void)
{
    __asm
    {
        NOP
    }
}

#pragma pop      // restore saved pragma state, back to arm state

int main(void)
{
    bar();
}
/* end of foo.c */

```

5.98.2 See also

- [--arm](#) on page 3-15
- [--thumb](#) on page 3-197
- [#pragma arm](#) on page 5-87.

5.99 #pragma unroll [(n)]

This pragma instructs the compiler to unroll a loop by n iterations.

Note

Both vectorized and nonvectorized loops can be unrolled using #pragma unroll [(n)]. That is, #pragma unroll [(n)] applies to both --vectorize and --no_vectorize.

5.99.1 Syntax

```
#pragma unroll
#pragma unroll (n)
```

Where:

n is an optional value indicating the number of iterations to unroll.

5.99.2 Default

If you do not specify a value for n , the compiler assumes #pragma unroll (4).

5.99.3 Usage

This pragma is only applicable if you are compiling with -O3 -Otime. When compiling with -O3 -Otime, the compiler automatically unrolls loops where it is beneficial to do so. You can use this pragma to ask the compiler to unroll a loop that has not been unrolled automatically.

Note

Use this pragma only when you have evidence, for example from --diag_warning=optimizations, that the compiler is not unrolling loops optimally by itself.

You cannot determine whether this pragma is having any effect unless you compile with --diag_warning=optimizations or examine the generated assembly code, or both.

5.99.4 Restrictions

This pragma can only take effect when you compile with -O3 -Otime. Even then, the use of this pragma is a *request* to the compiler to unroll a loop that has not been unrolled automatically. It does not guarantee that the loop is unrolled.

#pragma unroll [(n)] can be used only immediately before a **for** loop, a **while** loop, or a **do ... while** loop.

5.99.5 Example

```
void matrix_multiply(float ** __restrict dest, float ** __restrict src1,
                    float ** __restrict src2, unsigned int n)
{
    unsigned int i, j, k;
    for (i = 0; i < n; i++)
    {
        for (k = 0; k < n; k++)
        {
            float sum = 0.0f;
            /* #pragma unroll */
            for(j = 0; j < n; j++)
```

```

        sum += src1[i][j] * src2[j][k];
        dest[i][k] = sum;
    }
}

```

In this example, the compiler does not normally complete its loop analysis because `src2` is indexed as `src2[j][k]` but the loops are nested in the opposite order, that is, with `j` inside `k`. When `#pragma unroll` is uncommented in the example, the compiler proceeds to unroll the loop four times.

If the intention is to multiply a matrix that is not a multiple of four in size, for example an $n * n$ matrix, `#pragma unroll (m)` might be used instead, where m is some value so that n is an integral multiple of m .

5.99.6 See also

- [--diag_warning=optimizations](#) on page 3-76
- [-Onum](#) on page 3-156
- [-Otime](#) on page 3-161
- [--vectorize, --no_vectorize](#) on page 3-213
- [#pragma unroll_completely](#) on page 5-115
- [Loop unrolling in C code](#) on page 6-11 in *Using the Compiler*.

5.100 #pragma unroll_completely

This pragma instructs the compiler to completely unroll a loop. It has an effect only if the compiler can determine the number of iterations the loop has.

Note

Both vectorized and nonvectorized loops can be unrolled using #pragma unroll_completely. That is, #pragma unroll_completely applies to both --no_vectorize and --vectorize.

5.100.1 Usage

This pragma is only applicable if you are compiling with -O3 -Otime. When compiling with -O3 -Otime, the compiler automatically unrolls loops where it is beneficial to do so. You can use this pragma to ask the compiler to completely unroll a loop that has not automatically been unrolled completely.

Note

Use this #pragma only when you have evidence, for example from --diag_warning=optimizations, that the compiler is not unrolling loops optimally by itself.

You cannot determine whether this pragma is having any effect unless you compile with --diag_warning=optimizations or examine the generated assembly code, or both.

5.100.2 Restrictions

This pragma can only take effect when you compile with -O3 -Otime. Even then, the use of this pragma is a *request* to the compiler to unroll a loop that has not been unrolled automatically. It does not guarantee that the loop is unrolled.

#pragma unroll_completely can only be used immediately before a **for** loop, a **while** loop, or a **do ... while** loop.

Using #pragma unroll_completely on an outer loop can prevent vectorization. On the other hand, using #pragma unroll_completely on an inner loop might help in some cases.

5.100.3 See also

- [--diag_warning=optimizations on page 3-76](#)
- [-Onum on page 3-156](#)
- [-Otime on page 3-161](#)
- [--vectorize, --no_vectorize on page 3-213](#)
- [#pragma unroll \[\(n\)\] on page 5-113](#)
- [Loop unrolling in C code on page 6-11 in Using the Compiler.](#)

5.101 #pragma weak *symbol*, #pragma weak *symbol1* = *symbol2*

This pragma is a deprecated language extension to mark symbols as weak or to define weak aliases of symbols. It is an alternative to using the `__weak` keyword or the GCC `weak` and `alias` attributes.

5.101.1 Example

In the following example, `weak_fn` is declared as a weak alias of `__weak_fn`:

```
extern void weak_fn(int a);
#pragma weak weak_fn = __weak_fn

void __weak_fn(int a)
{
    ...
}
```

5.101.2 See also

- [__attribute__\(\(alias\)\) variable attribute](#) on page 5-71
- [__attribute__\(\(weak\)\) function attribute](#) on page 5-62
- [__attribute__\(\(weak\)\) variable attribute](#) on page 5-82
- [__weak](#) on page 5-27.

5.102 Instruction intrinsics

This topic describes instruction intrinsics for realizing ARM machine language instructions from C or C++ code. [Table 5-7](#) summarizes the available intrinsics.

Table 5-7 Instruction intrinsics supported by the ARM compiler

Instruction intrinsics		
<code>__breakpoint</code>	<code>__ldrt</code>	<code>__schedule_barrier</code>
<code>__cdp</code>	<code>__memory_changed</code>	<code>__semihost</code>
<code>__clrex</code>	<code>__nop</code>	<code>__sev</code>
<code>__clz</code>	<code>__pld</code>	<code>__sqrt</code>
<code>__current_pc</code>	<code>__pldw</code>	<code>__sqrtf</code>
<code>__current_sp</code>	<code>__pli</code>	<code>__ssat</code>
<code>__disable_fiq</code>	<code>__promise</code>	<code>__strex</code>
<code>__disable_irq</code>	<code>__qadd</code>	<code>__strexh</code>
<code>__enable_fiq</code>	<code>__qdbl</code>	<code>__str</code>
<code>__enable_irq</code>	<code>__qsub</code>	<code>__swp</code>
<code>__fabs</code>	<code>__rbit</code>	<code>__usat</code>
<code>__fabsf</code>	<code>__rev</code>	<code>__wfe</code>
<code>__force_stores</code>	<code>__return_address</code>	<code>__wfi</code>
<code>__ldrex</code>	<code>__ror</code>	<code>__yield</code>
<code>__ldrexh</code>		

See also [GNU built-in functions](#) on page 5-180.

5.103 `__breakpoint` intrinsic

This intrinsic inserts a BKPT instruction into the instruction stream generated by the compiler. It enables you to include a breakpoint instruction in your C or C++ code.

5.103.1 Syntax

```
void __breakpoint(int va1)
```

Where:

<code>va1</code>	is a compile-time constant integer whose range is:
0 ... 65535	if you are compiling source as ARM code
0 ... 255	if you are compiling source as Thumb code.

5.103.2 Errors

The compiler does not recognize the `__breakpoint` intrinsic when compiling for a target that does not support the BKPT instruction. The compiler generates either a warning or an error in this case.

The undefined instruction trap is taken if a BKPT instruction is executed on an architecture that does not support it.

5.103.3 Example

```
void func(void)
{
    ...
    __breakpoint(0xF02C);
    ...
}
```

5.103.4 See also

- [BKPT on page 3-49](#) in the *Assembler Reference*.

5.104 __cdp intrinsic

This intrinsic inserts a CDP or CDP2 instruction into the instruction stream generated by the compiler. It enables you to include coprocessor data operations in your C or C++ code.

———— **Note** ————

This intrinsic is intended for specialist expert use only.

5.104.1 Syntax

```
__cdp(unsigned int coproc, unsigned int ops, unsigned int regs)
```

Where:

<i>coproc</i>	Identifies the coprocessor the instruction is for. <i>coproc</i> must be an integer in the range 0 to 15.
<i>ops</i>	Is an encoding of the two opcodes for the CDP or CDP2 instruction, (<i>opcode1</i> <<3) <i>opcode2</i> , where: <ul style="list-style-type: none"> the first opcode, <i>opcode1</i>, is a 4-bit coprocessor-specific opcode the second opcode, <i>opcode2</i>, is an optional 3-bit coprocessor-specific opcode. Add 0x100 to <i>ops</i> to generate a CDP2 instruction.
<i>regs</i>	Is an encoding of the coprocessor registers, (<i>CRd</i> <<8) (<i>CRn</i> <<4) <i>CRm</i> , where <i>CRd</i> , <i>CRn</i> and <i>CRm</i> are the coprocessor registers for the CDP or CDP2 instruction.

5.104.2 Usage

The use of these instructions depends on the coprocessor. See your coprocessor documentation for more information.

5.104.3 Example

```
void copro_example()
{
    const unsigned int ops = 0xA3; // opcode1 = A, opcode2 = 3
    const unsigned int regs = 0xCDE; // reg1 = C, reg2 = D, reg3 = E
    __cdp(4,ops,regs); // coprocessor number 4
    // This intrinsic produces the instruction CDP p4,#0xa,c12,c13,c14,#3
}
```

5.104.4 See also

- [CDP and CDP2 on page 3-51](#) in the *Assembler Reference*.

5.105 `__clrex` intrinsic

This intrinsic inserts a CLREX instruction into the instruction stream generated by the compiler. It enables you to include a CLREX instruction in your C or C++ code.

5.105.1 Syntax

```
void __clrex(void)
```

5.105.2 Errors

The compiler does not recognize the `__clrex` intrinsic when compiling for a target that does not support the CLREX instruction. The compiler generates either a warning or an error in this case.

5.105.3 See also

- [CLREX](#) on page 3-53 in the *Assembler Reference*.

5.106 `__clz` intrinsic

This intrinsic inserts a CLZ instruction or an equivalent code sequence into the instruction stream generated by the compiler. It enables you to count the number of leading zeros of a data value in your C or C++ code.

5.106.1 Syntax

```
unsigned char __clz(unsigned int val)
```

Where:

`val` is an **unsigned int**.

5.106.2 Return value

The `__clz` intrinsic returns the number of leading zeros in `val`.

5.106.3 See also

- [Other built-in functions on page 5-182](#)
- [CLZ on page 3-54](#) in the *Assembler Reference*.

5.107 `__current_pc` intrinsic

This intrinsic enables you to determine the current value of the program counter at the point in your program where the intrinsic is used.

5.107.1 Syntax

```
unsigned int __current_pc(void)
```

5.107.2 Return value

The `__current_pc` intrinsic returns the current value of the program counter at the point in the program where the intrinsic is used.

5.107.3 See also

- [__current_sp intrinsic on page 5-123](#)
- [__return_address intrinsic on page 5-150](#)
- [Legacy inline assembler that accesses sp, lr, or pc on page 8-56](#) in *Using the Compiler*.

5.108 __current_sp intrinsic

This intrinsic returns the value of the stack pointer at the current point in your program.

5.108.1 Syntax

```
unsigned int __current_sp(void)
```

5.108.2 Return value

The `__current_sp` intrinsic returns the current value of the stack pointer at the point in the program where the intrinsic is used.

5.108.3 See also

- [Other built-in functions on page 5-182](#)
- [__current_pc intrinsic on page 5-122](#)
- [__return_address intrinsic on page 5-150](#)
- [Legacy inline assembler that accesses sp, lr, or pc on page 8-56](#) in *Using the Compiler*.

5.109 `__disable_fiq` intrinsic

This intrinsic disables FIQ interrupts.

———— Note ————

Typically, this intrinsic disables FIQ interrupts by setting the F-bit in the CPSR. However, for v7-M it sets the fault mask register (FAULTMASK). FIQ interrupts are not supported in v6-M.

5.109.1 Syntax

```
int __disable_fiq(void);
void __disable_fiq(void);
```

5.109.2 Usage

int __disable_fiq(void); disables fast interrupts and returns the value the FIQ interrupt mask has in the PSR prior to the disabling of interrupts.

void __disable_fiq(void); disables fast interrupts.

5.109.3 Return value

int __disable_fiq(void); returns the value the FIQ interrupt mask has in the PSR prior to the disabling of FIQ interrupts.

5.109.4 Restrictions

int __disable_fiq(void); is not supported when compiling with `--cpu=7`. This is because of the difference between the generic ARMv7 architecture and the ARMv7 A, R, and M-profiles in the exception handling model. This means that when you compile with `--cpu=7`, the compiler is unable to generate an instruction sequence that works on all ARMv7 processors and therefore **int __disable_fiq(void);** is not supported. You can use the **void __disable_fiq(void);** function prototype with `--cpu=7`.

The `__disable_fiq` intrinsic can only be executed in privileged modes, that is, in non-user modes. In User mode this intrinsic does not change the interrupt flags in the CPSR.

5.109.5 Example

```
void foo(void)
{
    int was_masked = __disable_fiq();
    /* ... */
    if (!was_masked)
        __enable_fiq();
}
```

5.109.6 See also

- [__enable_fiq intrinsic on page 5-127.](#)

5.110 __disable_irq intrinsic

This intrinsic disables IRQ interrupts.

———— Note ————

Typically, this intrinsic disables IRQ interrupts by setting the I-bit in the CPSR. However, for M-profile it sets the exception mask register (PRIMASK).

5.110.1 Syntax

```
int __disable_irq(void);
void __disable_irq(void);
```

5.110.2 Usage

int __disable_irq(void); disables interrupts and returns the value the IRQ interrupt mask has in the PSR prior to the disabling of interrupts.

void __disable_irq(void); disables interrupts.

5.110.3 Return value

int __disable_irq(void); returns the value the IRQ interrupt mask has in the PSR prior to the disabling of IRQ interrupts.

5.110.4 Example

```
void foo(void)
{
    int was_masked = __disable_irq();
    /* ... */
    if (!was_masked)
        __enable_irq();
}
```

5.110.5 Restrictions

int __disable_irq(void); is not supported when compiling with `--cpu=7`. This is because of the difference between the generic ARMv7 architecture and the ARMv7 A, R, and M-profiles in the exception handling model. This means that when you compile with `--cpu=7`, the compiler is unable to generate an instruction sequence that works on all ARMv7 processors and therefore **int __disable_irq(void);** is not supported. You can use the **void __disable_irq(void);** function prototype with `--cpu=7`.

The following example illustrates the difference between compiling for ARMv7-M and ARMv7-R:

```
/* test.c */ void DisableIrq(void)
{
    __disable_irq();
} int DisableIrq2(void)
{
    return __disable_irq();
}

armcc -c --cpu=Cortex-M3 -o m3.o test.c
```

```

DisableIrq
0x00000000: b672    r.    CPSID  i
0x00000002: 4770    pG    BX      lr
DisableIrq2
0x00000004: f3ef8010  .... MRS      r0,PRIMASK
0x00000008: f0000001  .... AND      r0,r0,#1
0x0000000c: b672    r.    CPSID  i
0x0000000e: 4770    pG    BX      lr

```

```
armcc -c --cpu=Cortex-R4 --thumb -o r4.o test.c
```

```

DisableIrq
0x00000000: b672    r.    CPSID  i
0x00000002: 4770    pG    BX      lr
DisableIrq2
0x00000004: f3ef8000  .... MRS      r0,APSR ; formerly CPSR
0x00000008: f0000080  .... AND      r0,r0,#0x80
0x0000000c: b672    r.    CPSID  i
0x0000000e: 4770    pG    BX      lr

```

In all cases, the `__disable_irq` intrinsic can only be executed in privileged modes, that is, in non-user modes. In User mode this intrinsic does not change the interrupt flags in the CPSR.

5.110.6 See also

- [__enable_irq intrinsic on page 5-128.](#)

5.111 `__enable_fiq` intrinsic

This intrinsic enables FIQ interrupts.

———— **Note** —————

Typically, this intrinsic enables FIQ interrupts by clearing the F-bit in the CPSR. However, for v7-M, it clears the fault mask register (FAULTMASK). FIQ interrupts are not supported in v6-M.

5.111.1 Syntax

```
void __enable_fiq(void)
```

5.111.2 Restrictions

The `__enable_fiq` intrinsic can only be executed in privileged modes, that is, in non-user modes. In User mode this intrinsic does not change the interrupt flags in the CPSR.

5.111.3 See also

- [__disable_fiq intrinsic](#) on page 5-124.

5.112 `__enable_irq` intrinsic

This intrinsic enables IRQ interrupts.

———— **Note** —————

Typically, this intrinsic enables IRQ interrupts by clearing the I-bit in the CPSR. However, for Cortex M-profile processors, it clears the exception mask register (PRIMASK).

5.112.1 Syntax

```
void __enable_irq(void)
```

5.112.2 Restrictions

The `__enable_irq` intrinsic can only be executed in privileged modes, that is, in non-user modes. In User mode this intrinsic does not change the interrupt flags in the CPSR.

5.112.3 See also

- [__disable_irq intrinsic](#) on page 5-125.

5.113 `__fabs` intrinsic

This intrinsic inserts a VABS instruction or an equivalent code sequence into the instruction stream generated by the compiler. It enables you to obtain the absolute value of a double-precision floating-point value from within your C or C++ code.

———— **Note** —————

The `__fabs` intrinsic is an analog of the standard C library function `fabs`. It differs from the standard library function in that a call to `__fabs` is guaranteed to be compiled into a single, inline, machine instruction on an ARM architecture-based processor equipped with a VFP coprocessor.

5.113.1 Syntax

```
double __fabs(double val)
```

Where:

`val` is a double-precision floating-point value.

5.113.2 Return value

The `__fabs` intrinsic returns the absolute value of `val` as a **double**.

5.113.3 See also

- [__fabsf intrinsic on page 5-130](#)
- [VABS, VNEG, and VSQRT on page 4-27](#) in the *Assembler Reference*.

5.114 `__fabsf` intrinsic

This intrinsic is a single-precision version of the `__fabs` intrinsic. It is functionally equivalent to `__fabs`, except that:

- it takes an argument of type **float** instead of an argument of type **double**
- it returns a **float** value instead of a **double** value.

5.114.1 Syntax

```
float __fabs(float val)
```

5.114.2 See also

- [__fabs intrinsic on page 5-129](#)
- [V{Q}ABS and V{Q}NEG on page 4-19](#) in the *Assembler Reference*.

5.115 `__force_stores` intrinsic

This intrinsic causes all variables that are visible outside the current function, such as variables that have pointers to them passed into or out of the function, to be written back to memory if they have been changed.

This intrinsic also acts as a scheduling barrier.

5.115.1 Syntax

```
void __force_stores(void)
```

5.115.2 See also

- [__memory_changed intrinsic](#) on page 5-137
- [__schedule_barrier intrinsic](#) on page 5-152.

5.116 __ldrex intrinsic

This intrinsic inserts an instruction of the form LDREX[*size*] into the instruction stream generated by the compiler. It enables you to load data from memory in your C or C++ code using an LDREX instruction. *size* in LDREX[*size*] is B for byte stores or H for halfword stores. If no size is specified, word stores are performed.

5.116.1 Syntax

```
unsigned int __ldrex(volatile void *ptr)
```

Where:

ptr points to the address of the data to be loaded from memory. To specify the type of the data to be loaded, cast the parameter to an appropriate pointer type.

Table 5-8 Access widths supported by the __ldrex intrinsic

Instruction	Size of data loaded	C cast
LDREXB	unsigned byte	(unsigned char *)
LDREXB	signed byte	(signed char *)
LDREXH	unsigned halfword	(unsigned short *)
LDREXH	signed halfword	(short *)
LDREX	word	(int *)

5.116.2 Return value

The __ldrex intrinsic returns the data loaded from the memory address pointed to by *ptr*.

5.116.3 Errors

The compiler does not recognize the __ldrex intrinsic when compiling for a target that does not support the LDREX instruction. The compiler generates either a warning or an error in this case.

The __ldrex intrinsic does not support access to doubleword data. The compiler generates an error if you specify an access width that is not supported.

5.116.4 Example

```
int foo(void)
{
    int loc = 0xff;
    return __ldrex((volatile char *)loc);
}
```

Compiling this code with the command-line option --cpu=6k produces

```
||foo|| PROC
MOV     r0,#0xff
LDREXB r0,[r0]
BX     lr
ENDP
```

5.116.5 See also

- [__ldrex intrinsic on page 5-134](#)

- [__strex intrinsic](#) on page 5-159
- [__strex intrinsic](#) on page 5-161
- [LDREX and STREX](#) on page 3-90 in the *Assembler Reference*.

5.117 __ldrex intrinsic

This intrinsic inserts an LDREXD instruction into the instruction stream generated by the compiler. It enables you to load data from memory in your C or C++ code using an LDREXD instruction. It supports access to doubleword data.

5.117.1 Syntax

```
unsigned long long __ldrex(volatile void *ptr)
```

Where:

ptr points to the address of the data to be loaded from memory. To specify the type of the data to be loaded, cast the parameter to an appropriate pointer type.

Table 5-9 Access widths supported by the __ldrex intrinsic

Instruction	Size of data loaded	C cast
LDREXD	unsigned long long	(unsigned long long *)
LDREXD	signed long long	(signed long long *)

5.117.2 Return value

The __ldrex intrinsic returns the data loaded from the memory address pointed to by *ptr*.

5.117.3 Errors

The compiler does not recognize the __ldrex intrinsic when compiling for a target that does not support the LDREXD instruction. The compiler generates either a warning or an error in this case.

The __ldrex intrinsic only supports access to doubleword data. The compiler generates an error if you specify an access width that is not supported.

5.117.4 See also

- [__ldrex intrinsic](#) on page 5-132
- [__strex intrinsic](#) on page 5-159
- [__strex intrinsic](#) on page 5-161
- [LDREX and STREX](#) on page 3-90 in the *Assembler Reference*.

5.118 __ldrt intrinsic

This intrinsic inserts an assembly language instruction of the form LDR{size}T into the instruction stream generated by the compiler. It enables you to load data from memory in your C or C++ code using an LDRT instruction.

5.118.1 Syntax

```
unsigned int __ldrt(const volatile void *ptr)
```

Where:

ptr Points to the address of the data to be loaded from memory. To specify the size of the data to be loaded, cast the parameter to an appropriate integral type.

Table 5-10 Access widths supported by the __ldrt intrinsic

Instruction ^a	Size of data loaded	C cast
LDRSBT	signed byte	(signed char *)
LDRBT	unsigned byte	(char *)
LDRSHT	signed halfword	(signed short int *)
LDRT	word	(int *)

a. Or equivalent.

5.118.2 Return value

The __ldrt intrinsic returns the data loaded from the memory address pointed to by *ptr*.

5.118.3 Errors

The compiler does not recognize the __ldrt intrinsic when compiling for a target that does not support the LDRT instruction. The compiler generates either a warning or an error in this case.

The __ldrt intrinsic does not support access to doubleword data. The compiler generates an error if you specify an access width that is not supported.

5.118.4 Example

```
int foo(void)
{
    int loc = 0xff;
    return __ldrt((const volatile int *)loc);
}
```

Compiling this code with the default options produces:

```
||foo|| PROC
MOV     r0,#0xff
LDRBT  r1,[r0],#0
MOV     r2,#0x100
LDRBT  r0,[r2],#0
ORR    r0,r1,r0,LSL #8
BX     lr
ENDP
```

5.118.5 See also

- [--thumb](#) on page 3-197
- [LDR and STR, unprivileged](#) on page 3-79 in the *ARM Assembler Reference*.

5.119 `__memory_changed` intrinsic

This intrinsic causes all variables that are visible outside the current function, such as variables that have pointers to them passed into or out of the function, to be written back to memory if they have been changed, and then to be read back from memory.

This intrinsic also acts as a scheduling barrier.

5.119.1 Syntax

```
void __memory_changed(void)
```

5.119.2 See also

- [__force_stores intrinsic on page 5-131](#)
- [__schedule_barrier intrinsic on page 5-152.](#)

5.120 `__nop` intrinsic

This intrinsic inserts a NOP instruction or an equivalent code sequence into the instruction stream.

The compiler does not optimize away the NOP instructions, except for normal unreachable code elimination. One NOP instruction is generated for each `__nop` intrinsic in the source.

ARMv6 and previous architectures do not have a NOP instruction, so the compiler generates a `MOV r0, r0` instruction instead.

In addition, `__nop` creates a special sequence point that prevents operations with side effects from moving past it under all circumstances. Normal sequence points allow operations with side effects past if they do not affect program behavior. Operations without side effects are not restricted by the intrinsic, and the compiler can move them past the sequence point. The `__schedule_barrier` intrinsic also creates this special sequence point, without inserting a NOP instruction.

Section 5.1.2.3 of the C standard defines operations with side effects as those that change the state of the execution environment. These operations:

- access volatile objects
- modify a memory location
- modify a file
- call a function that does any of the above.

In [Example 5-9](#), given this code, the compiler ensures that the read from the volatile variable `x` is enclosed between two NOP instructions.

Example 5-9 Volatile access

```
volatile int x;
int z;
int read_variable(int y)
{
    int i;
    int a = 0;
    __nop();
    a = x;
    __nop();
    return z + y;
}
```

If the `__nop` intrinsics are removed, and the compilation is performed at `-O3 -Otime` for `--cpu=Cortex-A8`, for example, then the compiler can schedule the read of the non-volatile variable `z` to be before the read of variable `x`.

In [Example 5-10](#), given this code, the compiler ensures that the write to variable `z` is enclosed between two NOP instructions.

Example 5-10 Modifying memory location

```
int x;
int z;
int write_variable(int y)
{
    int i;
    for (i = 0; i < 10; i++)
```

```

    {
        __nop();
        z = y;
        __nop();
        x += y;
    }
    return z;
}

```

In this case, if the `__nop` intrinsics are removed, then with `-O3 -Otime --cpu=Cortex-A8`, the compiler can fold away the loop.

In [Example 5-11](#), because `pure_func` has no side effects, the compiler can move the call to it outside of the loop. Still, the compiler ensures that the call to `func` is enclosed between two NOP instructions.

Example 5-11 Calling a pure function

```

int func(int x);
int pure_func(int x) __pure;
int read(int x)
{
    int i;
    int a=0;
    for (i=0; i<10; i++)
    {
        __nop();
        a += pure_func(x) + func(x);
        __nop();
    }
    return a;
}

```

Note

- You can use the `__schedule_barrier` intrinsic to insert a scheduling barrier without generating a NOP instruction.
- In the examples above, the compiler would treat `__schedule_barrier` in the same way as `__nop`.

5.120.1 Syntax

```
void __nop(void)
```

5.120.2 See also

- [__pure](#) on page 5-20
- [__sev intrinsic](#) on page 5-155
- [__schedule_barrier intrinsic](#) on page 5-152
- [__wfe intrinsic](#) on page 5-165
- [__wfi intrinsic](#) on page 5-166
- [__yield intrinsic](#) on page 5-167
- [NOP](#) on page 3-112 in the *Assembler Reference*
- [Generic intrinsics supported by the compiler on page 5-7](#) in *Using the Compiler*

- *Annex C, Sequence Points, C99 Standard*
 - [ISO C99 on page 2-5](#).

5.121 __pld intrinsic

This intrinsic inserts a data prefetch, for example PLD, into the instruction stream generated by the compiler. It enables you to signal to the memory system from your C or C++ program that a data load from an address is likely in the near future.

5.121.1 Syntax

```
void __pld(...)
```

Where:

... denotes any number of pointer or integer arguments specifying addresses of memory to prefetch.

5.121.2 Restrictions

If the target architecture does not support data prefetching, the compiler generates neither a PLD instruction nor a NOP instruction, but ignores the intrinsic.

5.121.3 Example

```
extern int data1;
extern int data2;
volatile int *interrupt = (volatile int *)0x8000;
volatile int *uart = (volatile int *)0x9000;
void get(void)
{
    __pld(data1, data2);
    while (!*interrupt);
    *uart = data1;        // trigger uart as soon as interrupt occurs
    *(uart+1) = data2;
}
```

5.121.4 See also

- [__pldw intrinsic](#) on page 5-142
- [__pli intrinsic](#) on page 5-143
- [PLD, PLDW, and PLI](#) on page 3-115 in the *Assembler Reference*.

5.122 `__pldw` intrinsic

This intrinsic inserts a PLDW instruction into the instruction stream generated by the compiler. It enables you to signal to the memory system from your C or C++ program that a data load from an address with an intention to write is likely in the near future.

5.122.1 Syntax

```
void __pldw(...)
```

Where:

... denotes any number of pointer or integer arguments specifying addresses of memory to prefetch.

5.122.2 Restrictions

If the target architecture does not support data prefetching, this intrinsic has no effect.

This intrinsic only takes effect in ARMv7 architectures and above that provide Multiprocessing Extensions. That is, when the predefined macro `__TARGET_FEATURE_MULTIPROCESSING` is defined.

5.122.3 Example

```
void foo(int *bar)
{
    __pldw(bar);
}
```

5.122.4 See also

- [Predefined macros](#) on page 5-183
- [__pld intrinsic](#) on page 5-141
- [__pli intrinsic](#) on page 5-143
- [PLD, PLDW, and PLI](#) on page 3-115 in the *Assembler Reference*.

5.123 `__pli` intrinsic

This intrinsic inserts an instruction prefetch, for example PLI, into the instruction stream generated by the compiler. It enables you to signal to the memory system from your C or C++ program that an instruction load from an address is likely in the near future.

5.123.1 Syntax

```
void __pli(...)
```

Where:

... denotes any number of pointer or integer arguments specifying addresses of instructions to prefetch.

5.123.2 Restrictions

If the target architecture does not support instruction prefetching, the compiler generates neither a PLI instruction nor a NOP instruction, but ignores the intrinsic.

5.123.3 See also

- [__pld intrinsic on page 5-141](#)
- [__pldw intrinsic on page 5-142](#)
- [PLD, PLDW, and PLI on page 3-115](#) in the *Assembler Reference*.

5.124 `__promise` intrinsic

This intrinsic represents a promise you make to the compiler that a given expression always has a nonzero value. This enables the compiler to perform more aggressive optimization when vectorizing code.

5.124.1 Syntax

```
void __promise(expr)
```

Where *expr* is an expression that evaluates to nonzero.

5.124.2 Usage

`__promise(expr)` is similar but complementary to `assert(expr)`. Unlike `assert(expr)`, `__promise(expr)` is effective when `NDEBUG` is defined.

If assertions are enabled (by including `assert.h` and not defining `NDEBUG`) then the promise is checked at runtime by evaluating *expr* as part of `assert(expr)`.

5.124.3 See also

- [Indicating loop iteration counts to the compiler with `__promise\(expr\)` on page 4-25 in *Using the Compiler*.](#)

5.125 `__qadd` intrinsic

This intrinsic inserts a QADD instruction into the instruction stream generated by the compiler. It enables you to obtain the saturating add of two integers from within your C or C++ code.

———— **Note** —————

The compiler might optimize your code when it detects opportunity to do so, using equivalent instructions from the same family to produce fewer instructions.

5.125.1 Syntax

```
int __qadd(int va11, int va12)
```

Where:

`va11` is the first summand of the saturating add operation
`va12` is the second summand of the saturating add operation.

5.125.2 Return value

The `__qadd` intrinsic returns the saturating add of `va11` and `va12`.

5.125.3 Restriction

This intrinsic is only available on targets that have the QADD instruction.

5.125.4 See also

- [__qdbl intrinsic on page 5-146](#)
- [__qsub intrinsic on page 5-147](#)
- [QADD, QSUB, QDADD, and QDSUB on page 3-119](#) in the *Assembler Reference*.

5.126 `__qdbl` intrinsic

This intrinsic inserts instructions equivalent to the saturating add of an integer with itself into the instruction stream generated by the compiler. It enables you to obtain the saturating double of an integer from within your C or C++ code.

5.126.1 Syntax

```
int __qdbl(int va1)
```

Where:

`va1` is the data value to be doubled.

5.126.2 Return value

The `__qdbl` intrinsic returns the saturating add of `va1` with itself, or equivalently, `__qadd(va1, va1)`.

5.126.3 See also

- [__qadd intrinsic on page 5-145](#).

5.127 __qsub intrinsic

This intrinsic inserts a QSUB instruction or an equivalent code sequence into the instruction stream generated by the compiler. It enables you to obtain the saturating subtraction of two integers from within your C or C++ code.

5.127.1 Syntax

```
int __qsub(int va11, int va12)
```

Where:

va11 is the minuend of the saturating subtraction operation

va12 is the subtrahend of the saturating subtraction operation.

5.127.2 Return value

The __qsub intrinsic returns the saturating subtraction of *va11* and *va12*.

5.127.3 See also

- [__qadd intrinsic on page 5-145](#)
- [QADD, QSUB, QDADD, and QDSUB on page 3-119](#) in the *Assembler Reference*.

5.128 `__rbit` intrinsic

This intrinsic inserts an RBIT instruction into the instruction stream generated by the compiler. It enables you to reverse the bit order in a 32-bit word from within your C or C++ code.

5.128.1 Syntax

```
unsigned int __rbit(unsigned int val)
```

Where:

`val` is the data value whose bit order is to be reversed.

5.128.2 Return value

The `__rbit` intrinsic returns the value obtained from `val` by reversing its bit order.

5.128.3 See also

- [REV](#), [REV16](#), [REVSH](#), and [RBIT](#) on page 3-121 in the *Assembler Reference*.

5.129 `__rev` intrinsic

This intrinsic inserts a REV instruction or an equivalent code sequence into the instruction stream generated by the compiler. It enables you to convert a 32-bit big-endian data value into a little-endian data value, or a 32-bit little-endian data value into a big-endian data value from within your C or C++ code.

———— **Note** —————

The `__rev` intrinsic is available irrespective of the target processor or architecture you are compiling for. However, if the REV instruction is not available on the target, the compiler compensates with an alternative code sequence that could increase the number of instructions, effectively expanding the intrinsic into a function.

———— **Note** —————

The compiler introduces REV automatically when it recognizes certain expressions.

5.129.1 Syntax

```
unsigned int __rev(unsigned int val)
```

Where:

`val` is an **unsigned int**.

5.129.2 Return value

The `__rev` intrinsic returns the value obtained from `val` by reversing its byte order.

5.129.3 See also

- [REV, REV16, REVSH, and RBIT on page 3-121](#) in the *Assembler Reference*.

5.130 `__return_address` intrinsic

This intrinsic returns the return address of the current function.

5.130.1 Syntax

```
unsigned int __return_address(void)
```

5.130.2 Return value

The `__return_address` intrinsic returns the value of the link register that is used in returning from the current function.

5.130.3 Restrictions

The `__return_address` intrinsic does *not* affect the ability of the compiler to perform optimizations such as inlining, tailcalling, and code sharing. Where optimizations are made, the value returned by `__return_address` reflects the optimizations performed:

No optimization

When no optimizations are performed, the value returned by `__return_address` from within a function `foo` is the return address of `foo`.

Inline optimization

If a function `foo` is inlined into a function `bar` then the value returned by `__return_address` from within `foo` is the return address of `bar`.

Tail-call optimization

If a function `foo` is tail-called from a function `bar` then the value returned by `__return_address` from within `foo` is the return address of `bar`.

5.130.4 See also

- [Other built-in functions on page 5-182](#)
- [__current_pc intrinsic on page 5-122](#)
- [__current_sp intrinsic on page 5-123](#)
- [Legacy inline assembler that accesses sp, lr, or pc on page 8-56](#) in the *Compiler Reference*.

5.131 `__ror` intrinsic

This intrinsic inserts a ROR instruction or operand rotation into the instruction stream generated by the compiler. It enables you to rotate a value right by a specified number of places from within your C or C++ code.

———— **Note** —————

The compiler introduces ROR automatically when it recognizes certain expressions.

5.131.1 Syntax

```
unsigned int __ror(unsigned int val, unsigned int shift)
```

Where:

val is the value to be shifted right

shift is a constant shift in the range 1-31.

5.131.2 Return value

The `__ror` intrinsic returns the value of *val* rotated right by *shift* number of places.

5.131.3 See also

- [ASR, LSL, LSR, ROR, and RRX on page 3-42](#) in the *Assembler Reference*.

5.132 `__schedule_barrier` intrinsic

This intrinsic creates a special sequence point that prevents operations with side effects from moving past it under all circumstances. Normal sequence points allow operations with side effects past if they do not affect program behavior. Operations without side effects are not restricted by the intrinsic, and the compiler can move them past the sequence point.

Unlike the `__force_stores` intrinsic, the `__schedule_barrier` intrinsic does not cause memory to be updated. The `__schedule_barrier` intrinsic is similar to the `__nop` intrinsic, only differing in that it does not generate a NOP instruction.

5.132.1 Syntax

```
void __schedule_barrier(void)
```

5.132.2 See also

- [__force_stores intrinsic](#) on page 5-131
- [__nop intrinsic](#) on page 5-138.

5.133 __semihost intrinsic

This intrinsic inserts an SVC or BKPT instruction into the instruction stream generated by the compiler. It enables you to make semihosting calls from C or C++ that are independent of the target architecture.

5.133.1 Syntax

```
int __semihost(int val, const void *ptr)
```

Where:

val Is the request code for the semihosting request.

See [Chapter 8 Semihosting](#) in *Developing Software for ARM Processors* for more information.

ptr Is a pointer to an argument/result block.

See [Chapter 8 Semihosting](#) in *Developing Software for ARM Processors* for more information.

5.133.2 Return value

See [Chapter 8 Semihosting](#) in *Developing Software for ARM Processors* for more information on the results of semihosting calls.

5.133.3 Usage

Use this intrinsic from C or C++ to generate the appropriate semihosting call for your target and instruction set:

SVC 0x123456 In ARM state for all architectures.

SVC 0xAB In Thumb state, excluding M-profile architectures. This behavior is not guaranteed on *all* debug targets from ARM or from third parties.

BKPT 0xAB For M-profile architectures (Thumb only).

5.133.4 Restrictions

ARM processors prior to ARMv7 use SVC instructions to make semihosting calls. However, if you are compiling for a Cortex M-profile processor, semihosting is implemented using the BKPT instruction.

5.133.5 Example

```
char buffer[100];
...
void foo(void)
{
    __semihost(0x01, (const void *)buf); // equivalent in thumb state to
                                        // int __svc(0xAB) my_svc(int, int *);
                                        // result = my_svc(0x1, &buffer);
}

```

Compiling this code with the option `--thumb` generates:

```
||foo|| PROC
...
LDR    r1, |L1.12|
```

```

        MOVS    r0,#1
        SVC    #0xab
        ...
|L1.12|
        ...
buffer
%      400

```

5.133.6 See also

- [--cpu=list](#) on page 3-48
- [--thumb](#) on page 3-197
- [__svc](#) on page 5-23
- [BKPT](#) on page 3-49 in the *Assembler Reference*
- [SVC](#) on page 3-154 in the *Assembler Reference*
- [Chapter 8 Semihosting](#) in *Developing Software for ARM Processors*.

5.134 `__sev` intrinsic

This intrinsic inserts a SEV instruction into the instruction stream generated by the compiler.

In some architectures, for example the v6T2 architecture, the SEV instruction executes as a NOP instruction.

5.134.1 Syntax

```
void __sev(void)
```

5.134.2 Errors

The compiler does not recognize the `__sev` intrinsic when compiling for a target that does not support the SEV instruction. The compiler generates either a warning or an error in this case.

5.134.3 See also

- [__nop intrinsic on page 5-138](#)
- [__wfe intrinsic on page 5-165](#)
- [__wfi intrinsic on page 5-166](#)
- [__yield intrinsic on page 5-167](#)
- [NOP on page 3-112 in the *Assembler Reference*.](#)

5.135 `__sqrt` intrinsic

This intrinsic inserts a VFP VSQRT instruction into the instruction stream generated by the compiler. It enables you to obtain the square root of a double-precision floating-point value from within your C or C++ code.

———— **Note** —————

The `__sqrt` intrinsic is an analog of the standard C library function `sqrt`. It differs from the standard library function in that a call to `__sqrt` is guaranteed to be compiled into a single, inline, machine instruction on an ARM architecture-based processor equipped with a VFP coprocessor.

5.135.1 Syntax

```
double __sqrt(double va1)
```

Where:

`va1` is a double-precision floating-point value.

5.135.2 Return value

The `__sqrt` intrinsic returns the square root of `va1` as a **double**.

5.135.3 Errors

The compiler does not recognize the `__sqrt` intrinsic when compiling for a target that is not equipped with a VFP coprocessor. The compiler generates either a warning or an error in this case.

5.135.4 See also

- [__sqrtf intrinsic on page 5-157](#)
- [VABS, VNEG, and VSQRT on page 4-27](#) in the *Assembler Reference*.

5.136 `__sqrtf` intrinsic

This intrinsic is a single-precision version of the `__sqrtf` intrinsic. It is functionally equivalent to `__sqrt`, except that:

- it takes an argument of type **float** instead of an argument of type **double**
- it returns a **float** value instead of a **double** value.

5.136.1 See also

- [__sqrt intrinsic on page 5-156](#)
- [VABS, VNEG, and VSQRT on page 4-27](#) in the *Assembler Reference*.

5.137 `__ssat` intrinsic

This intrinsic inserts an SSAT instruction into the instruction stream generated by the compiler. It enables you to saturate a signed value from within your C or C++ code.

5.137.1 Syntax

```
int __ssat(int val, unsigned int sat)
```

Where:

`val` Is the value to be saturated.
`sat` Is the bit position to saturate to.
`sat` must be in the range 1 to 32.

5.137.2 Return value

The `__ssat` intrinsic returns `val` saturated to the signed range $-2^{sat-1} \leq x \leq 2^{sat-1} - 1$.

5.137.3 Errors

The compiler does not recognize the `__ssat` intrinsic when compiling for a target that does not support the SSAT instruction. The compiler generates either a warning or an error in this case.

5.137.4 See also

- [__usat intrinsic on page 5-164](#)
- [SSAT and USAT on page 3-148](#) in the *Assembler Reference*.

5.138 __strex intrinsic

This intrinsic inserts an instruction of the form STREX[size] into the instruction stream generated by the compiler. It enables you to use an STREX instruction in your C or C++ code to store data to memory.

5.138.1 Syntax

```
int __strex(unsigned int val, volatile void *ptr)
```

Where:

val is the value to be written to memory.

ptr points to the address of the data to be written to in memory. To specify the size of the data to be written, cast the parameter to an appropriate integral type.

Table 5-11 Access widths supported by the __strex intrinsic

Instruction	Size of data stored	C cast
STREXB	unsigned byte	(char *)
STREXH	unsigned halfword	(short *)
STREX	word	(int *)

5.138.2 Return value

The __strex intrinsic returns:

0 if the STREX instruction succeeds
 1 if the STREX instruction is locked out.

5.138.3 Errors

The compiler does not recognize the __strex intrinsic when compiling for a target that does not support the STREX instruction. The compiler generates either a warning or an error in this case.

The __strex intrinsic does not support access to doubleword data. The compiler generates an error if you specify an access width that is not supported.

5.138.4 Example

```
int foo(void)
{
    int loc=0xff;
    return(!__strex(0x20, (volatile char *)loc));
}
```

Compiling this code with the command-line option --cpu=6k produces

```
||foo|| PROC
    MOV     r0,#0xff
    MOV     r2,#0x20
    STREXB  r1,r2,[r0]
    RSBS   r0,r1,#1
    MOVCC  r0,#0
    BX     lr
ENDP
```

5.138.5 See also

- [__ldrex intrinsic](#) on page 5-132
- [__ldrex intrinsic](#) on page 5-134
- [__strex intrinsic](#) on page 5-161
- [LDREX and STREX](#) on page 3-90 in the *Assembler Reference*.

5.139 __strexld intrinsic

This intrinsic inserts an STREXD instruction into the instruction stream generated by the compiler. It enables you to use an STREXD instruction in your C or C++ code to store data to memory. It supports exclusive stores of doubleword data to memory.

5.139.1 Syntax

```
int __strexld(unsigned long long val, volatile void *ptr)
```

Where:

val is the value to be written to memory.

ptr points to the address of the data to be written to in memory. To specify the size of the data to be written, cast the parameter to an appropriate integral type.

Table 5-12 Access widths supported by the __strexld intrinsic

Instruction	Size of data stored	C cast
STREXD	unsigned long long	(unsigned long long *)
STREXD	signed long long	(signed long long *)

5.139.2 Return value

The __strexld intrinsic returns:

0 if the STREXD instruction succeeds
 1 if the STREXD instruction is locked out.

5.139.3 Errors

The compiler does not recognize the __strexld intrinsic when compiling for a target that does not support the STREXD instruction. The compiler generates either a warning or an error in this case.

The __strexld intrinsic only supports access to doubleword data. The compiler generates an error if you specify an access width that is not supported.

5.139.4 See also

- [__ldrex intrinsic on page 5-132](#)
- [__ldrexld intrinsic on page 5-134](#)
- [__strex intrinsic on page 5-159](#)
- [LDREX and STREX on page 3-90](#) in the *Assembler Reference*.

5.140 __strt intrinsic

This intrinsic inserts an assembly language instruction of the form STR{size}T into the instruction stream generated by the compiler. It enables you to store data to memory in your C or C++ code using an STRT instruction.

5.140.1 Syntax

```
void __strt(unsigned int val, volatile void *ptr)
```

Where:

val Is the value to be written to memory.

ptr Points to the address of the data to be written to in memory. To specify the size of the data to be written, cast the parameter to an appropriate integral type.

Table 5-13 Access widths supported by the __strt intrinsic

Instruction	Size of data loaded	C cast
STRBT	unsigned byte	(char *)
STRHT	unsigned halfword	(short int *)
STRT	word	(int *)

5.140.2 Errors

The compiler does not recognize the __strt intrinsic when compiling for a target that does not support the STRT instruction. The compiler generates either a warning or an error in this case.

The __strt intrinsic does not support access either to signed data or to doubleword data. The compiler generates an error if you specify an access width that is not supported.

5.140.3 Example

```
void foo(void)
{
    int loc=0xff;
    __strt(0x20, (volatile char *)loc);
}
```

Compiling this code produces:

```
||foo|| PROC
MOV     r0,#0xff
MOV     r1,#0x20
STRBT  r1,[r0],#0
BX      lr
ENDP
```

5.140.4 See also

- [--thumb](#) on page 3-197
- [LDR and STR, unprivileged](#) on page 3-79 in the *Assembler Reference*.

5.141 `__swp` intrinsic

This intrinsic inserts a `SWP{size}` instruction into the instruction stream generated by the compiler. It enables you to swap data between memory locations from your C or C++ code.

———— **Note** ————

The use of `SWP` and `SWPB` is deprecated in ARMv6 and above.

5.141.1 Syntax

```
unsigned int __swp(unsigned int val, volatile void *ptr)
```

Where:

`val` Is the data value to be written to memory.

`ptr` Points to the address of the data to be written to in memory. To specify the size of the data to be written, cast the parameter to an appropriate integral type.

Table 5-14 Access widths supported by the `__swp` intrinsic

Instruction	Size of data loaded	C cast
SWPB	unsigned byte	(char *)
SWP	word	(int *)

5.141.2 Return value

The `__swp` intrinsic returns the data value that previously, is in the memory address pointed to by `ptr`, before this value is overwritten by `val`.

5.141.3 Example

```
int foo(void)
{
    int loc=0xff;
    return(__swp(0x20, (volatile int *)loc));
}
```

Compiling this code produces

```
||foo|| PROC
MOV     r1, #0xff
MOV     r0, #0x20
SWP     r0, r0, [r1]
BX      lr
ENDP
```

5.141.4 See also

- [SWP and SWPB on page 3-155](#) in the *Assembler Reference*.

5.142 __usat intrinsic

This intrinsic inserts a USAT instruction into the instruction stream generated by the compiler. It enables you to saturate an unsigned value from within your C or C++ code.

5.142.1 Syntax

```
int __usat(unsigned int val, unsigned int sat)
```

Where:

val Is the value to be saturated.
sat Is the bit position to saturate to.
sat must be in the range 0 to 31.

5.142.2 Return value

The __usat intrinsic returns *val* saturated to the unsigned range $0 \leq x \leq 2^{sat} - 1$.

5.142.3 Errors

The compiler does not recognize the __usat intrinsic when compiling for a target that does not support the USAT instruction. The compiler generates either a warning or an error in this case.

5.142.4 See also

- [__ssat intrinsic on page 5-158](#)
- [SSAT and USAT on page 3-148](#) in the *Assembler Reference*.

5.143 `__wfe` intrinsic

This intrinsic inserts a WFE instruction into the instruction stream generated by the compiler.

In some architectures, for example the v6T2 architecture, the WFE instruction executes as a NOP instruction.

5.143.1 Syntax

```
void __wfe(void)
```

5.143.2 Errors

The compiler does not recognize the `__wfe` intrinsic when compiling for a target that does not support the WFE instruction. The compiler generates either a warning or an error in this case.

5.143.3 See also

- [__wfi intrinsic on page 5-166](#)
- [__nop intrinsic on page 5-138](#)
- [__sev intrinsic on page 5-155](#)
- [__yield intrinsic on page 5-167](#)
- [NOP on page 3-112 in the *Assembler Reference*.](#)

5.144 `__wfi` intrinsic

This intrinsic inserts a WFI instruction into the instruction stream generated by the compiler.

In some architectures, for example the v6T2 architecture, the WFI instruction executes as a NOP instruction.

5.144.1 Syntax

```
void __wfi(void)
```

5.144.2 Errors

The compiler does not recognize the `__wfi` intrinsic when compiling for a target that does not support the WFI instruction. The compiler generates either a warning or an error in this case.

5.144.3 See also

- [__yield intrinsic](#) on page 5-167
- [__nop intrinsic](#) on page 5-138
- [__sev intrinsic](#) on page 5-155
- [__wfe intrinsic](#) on page 5-165
- [NOP](#) on page 3-112 in the *Assembler Reference*.

5.145 `__yield` intrinsic

This intrinsic inserts a YIELD instruction into the instruction stream generated by the compiler.

In some architectures, for example the v6T2 architecture, the YIELD instruction executes as a NOP instruction.

5.145.1 Syntax

```
void __yield(void)
```

5.145.2 Errors

The compiler does not recognize the `__yield` intrinsic when compiling for a target that does not support the YIELD instruction. The compiler generates either a warning or an error in this case.

5.145.3 See also

- [__nop intrinsic on page 5-138](#)
- [__sev intrinsic on page 5-155](#)
- [__wfe intrinsic on page 5-165](#)
- [__wfi intrinsic on page 5-166](#)
- [NOP on page 3-112 in the *Assembler Reference*.](#)

5.146 ARMv6 SIMD intrinsics

The ARM Architecture v6 Instruction Set Architecture adds many *Single Instruction Multiple Data* (SIMD) instructions to ARMv6 for the efficient software implementation of high-performance media applications.

The ARM compiler supports intrinsics that map to the ARMv6 SIMD instructions. These intrinsics are available when compiling your code for an ARMv6 architecture or processor. If the chosen architecture does not support the ARMv6 SIMD instructions, compilation generates a warning and subsequent linkage fails with an undefined symbol reference.

Note

Each ARMv6 SIMD intrinsic is guaranteed to be compiled into a single, inline, machine instruction for an ARM v6 architecture or processor. However, the compiler might use optimized forms of underlying instructions when it detects opportunities to do so.

The ARMv6 SIMD instructions can set the GE[3:0] bits in the *Application Program Status Register* (APSR). Some SIMD instructions update these flags to indicate the *greater than or equal to* status of each 8 or 16-bit slice of an SIMD operation.

The ARM compiler treats the GE[3:0] bits as a global variable. To access these bits from within your C or C++ program, either:

- access bits 16-19 of the APSR through a named register variable
- use the `__sel` intrinsic to control a SEL instruction.

5.146.1 See also

Reference

- [Appendix A ARMv6 SIMD Instruction Intrinsics](#) on page A-1
- [Named register variables](#) on page 5-176
- [ARM registers](#) on page 3-9 in *Using the Assembler*
- [SEL](#) on page 3-127 in the *Assembler Reference*
- [Chapter 9 NEON and VFP Programming](#) in the *Using the Assembler*.

5.147 ETSI basic operations

The compilation tools support the original ETSI family of basic operations described in the ETSI G.729 recommendation *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*.

To make use of the ETSI basic operations in your own code, include the standard header file `dspfns.h`. The intrinsics supplied in `dspfns.h` are listed in [Table 5-15](#).

Table 5-15 ETSI basic operations supported by the ARM compilation tools

Intrinsics				
<code>abs_s</code>	<code>L_add_c</code>	<code>L_mult</code>	<code>L_sub_c</code>	<code>norm_l</code>
<code>add</code>	<code>L_deposit_h</code>	<code>L_negate</code>	<code>mac_r</code>	<code>round</code>
<code>div_s</code>	<code>L_deposit_l</code>	<code>L_sat</code>	<code>msu_r</code>	<code>saturate</code>
<code>extract_h</code>	<code>L_mac</code>	<code>L_shl</code>	<code>mult</code>	<code>shl</code>
<code>extract_l</code>	<code>L_macNs</code>	<code>L_shr</code>	<code>mult_r</code>	<code>shr</code>
<code>L_abs</code>	<code>L_msu</code>	<code>L_shr_r</code>	<code>negate</code>	<code>shr_r</code>
<code>L_add</code>	<code>L_msuNs</code>	<code>L_sub</code>	<code>norm_s</code>	<code>sub</code>

The header file `dspfns.h` also exposes certain status flags as global variables for use in your C or C++ programs. The status flags exposed by `dspfns.h` are listed in [Table 5-16](#).

Table 5-16 ETSI status flags exposed in the ARM compilation tools

Status flag	Description
Overflow	Overflow status flag. Generally, saturating functions have a sticky effect on overflow.
Carry	Carry status flag.

5.147.1 Example

```
#include <limits.h>
#include <stdint.h>
#include <dspfns.h> // include ETSI basic operations
int32_t C_L_add(int32_t a, int32_t b)
{
    int32_t c = a + b;
    if (((a ^ b) & INT_MIN) == 0)
    {
        if ((c ^ a) & INT_MIN)
        {
            c = (a < 0) ? INT_MIN : INT_MAX;
        }
    }
    return c;
}
__asm int32_t asm_L_add(int32_t a, int32_t b)
{
    qadd r0, r0, r1
    bx lr
}
int32_t foo(int32_t a, int32_t b)
```

```

{
  int32_t c, d, e, f;
  Overflow = 0;          // set global overflow flag
  c = C_L_add(a, b);    // C saturating add
  d = asm_L_add(a, b);  // assembly language saturating add
  e = __qadd(a, b);     // ARM intrinsic saturating add
  f = L_add(a, b);      // ETSI saturating add
  return Overflow ? -1 : c == d == e == f; // returns 1, unless overflow
}

```

5.147.2 See also

- the header file `dspfns.h` for definitions of the ETSI basic operations as a combination of C code and intrinsics
- [European Telecommunications Standards Institute \(ETSI\) basic operations on page 5-12](#) in *Using the Compiler*
- ETSI Recommendation G.191: *Software tools for speech and audio coding standardization*
- *ITU-T Software Tool Library 2005 User's manual*, included as part of ETSI Recommendation G.191
- ETSI Recommendation G.723.1 : *Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*
- ETSI Recommendation G.729: *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*.

5.148 C55x intrinsics

The ARM compiler supports the emulation of selected TI C55x compiler intrinsics.

To make use of the TI C55x intrinsics in your own code, include the standard header file `c55x.h`. The intrinsics supplied in `c55x.h` are listed in [Table 5-17](#).

Table 5-17 TI C55x intrinsics supported by the compilation tools

Intrinsics			
<code>_a_lsadd</code>	<code>_a_sadd</code>	<code>_a_smac</code>	<code>_a_smacr</code>
<code>_a_smas</code>	<code>_a_smasr</code>	<code>_abss</code>	<code>_count</code>
<code>_divs</code>	<code>_labss</code>	<code>_lmax</code>	<code>_lmin</code>
<code>_lmpy</code>	<code>_lmpysu</code>	<code>_lmpyu</code>	<code>_lnorm</code>
<code>_lsadd</code>	<code>_lsat</code>	<code>_lshl</code>	<code>_shrs</code>
<code>_lsmpy</code>	<code>_lsmpyi</code>	<code>_lsmpyr</code>	<code>_lsmpyu</code>
<code>_lsmypsui</code>	<code>_lsmpyu</code>	<code>_lsmpyui</code>	<code>_lsneg</code>
<code>_lssh1</code>	<code>_lssub</code>	<code>_max</code>	<code>_min</code>
<code>_norm</code>	<code>_rnd</code>	<code>_round</code>	<code>_roundn</code>
<code>_sadd</code>	<code>_shl</code>	<code>_shrs</code>	<code>_smac</code>
<code>_smaci</code>	<code>_smacr</code>	<code>_smacsu</code>	<code>_smacsui</code>
<code>_smas</code>	<code>_masi</code>	<code>_masr</code>	<code>_massu</code>
<code>_massui</code>	<code>_smpy</code>	<code>_sneg</code>	<code>_sround</code>
<code>_sroundn</code>	<code>_ssh1</code>	<code>_ssub</code>	-

5.148.1 Restrictions

The C55x intrinsics are only supported on targets that support the `__qadd`, `__qdbl`, and `__qsub` intrinsics. Otherwise, no error message is generated, instead the compiler silently generates a call to a corresponding function `__qadd`, `__qdbl`, or `__qsub`.

5.148.2 Example

```
#include <limits.h>
#include <stdint.h>
#include <c55x.h> // include TI C55x intrinsics
__asm int32_t asm_lsadd(int32_t a, int32_t b)
{
    qadd r0, r0, r1
    bx lr}
int32_t foo(int32_t a, int32_t b)
{
    int32_t c, d, e;
    c = asm_lsadd(a, b); // assembly language saturating add
    d = __qadd(a, b); // ARM intrinsic saturating add
    e = _lsadd(a, b); // TI C55x saturating add
    return c == d == e; // returns 1
}
```


5.148.3 See also

- the header file `c55x.h` for more information on the ARM implementation of the C55x intrinsics
- publications providing information about TI compiler intrinsics are available from Texas Instruments at <http://www.ti.com>.

5.149 VFP status intrinsic

The compiler provides an intrinsic for reading the *Floating Point and Status Control Register* (FPSCR).

———— **Note** —————

ARM recommends using a named register variable as an alternative method of reading this register. This provides a more efficient method of access than using the intrinsic.

See:

- [__vfp_status intrinsic](#) on page 5-174
- [Named register variables](#) on page 5-176.

5.150 `__vfp_status` intrinsic

This intrinsic reads or modifies the FPSCR.

5.150.1 Syntax

```
unsigned int __vfp_status(unsigned int mask, unsigned int flags);
```

5.150.2 Usage

Use this intrinsic to read or modify the flags in FPSCR.

The intrinsic returns the value of FPSCR, unmodified, if *mask* and *flags* are 0.

You can clear, set, or toggle individual flags in FPSCR using the bits in *mask* and *flags*, as shown in [Table 5-18](#). The intrinsic returns the modified value of FPSCR if *mask* and *flags* are not both 0.

Table 5-18 Modifying the FPSCR flags

<i>mask</i> bit	<i>flags</i> bit	Effect on FPSCR flag
0	0	Does not modify the flag
0	1	Toggles the flag
1	1	Sets the flag
1	0	Clears the flag

———— **Note** —————

If you want to read or modify only the exception flags in FPSCR, then ARM recommends that you use the standard C99 features in `<fenv.h>`.

5.150.3 Errors

The compiler generates an error if you attempt to use this intrinsic when compiling for a target that does not have VFP.

5.150.4 See also

- [FPSCR, the floating-point status and control register on page 9-24](#) in *Using the Assembler*
- [<fenv.h> floating-point environment access in C99 on page 6-105](#) in *Using the Compiler*.

5.151 Fused Multiply Add (FMA) intrinsics

These intrinsics perform the following calculation, incurring only a single rounding step:

$$result = a \times b + c$$

Performing the calculation with a single rounding step, rather than multiplying and then adding with two roundings, can result in a better degree of accuracy.

Declared in `math.h`, the FMA intrinsics are:

```
double fma(double a, double b, double c);
float fmaf(float a, float b, float c);
long double fmal(long double a, long double b, long double c);
```

———— **Note** ————

- These intrinsics are only available in C99 mode.
 - They are only supported for the Cortex-A5 and Cortex-M4 processors.
 - If compiling for the Cortex-M4 processor, only `fmaf()` is available.
-

5.152 Named register variables

The compiler enables you to access registers of an ARM architecture-based processor or coprocessor using named register variables.

5.152.1 Syntax

```
register type var-name __asm(reg);
```

Where:

type is the type of the named register variable.

Any type of the same size as the register being named can be used in the declaration of a named register variable. The type can be a structure, but bitfield layout is sensitive to endianness.

var-name is the name of the named register variable.

reg is a character string denoting the name of a register on an ARM architecture-based processor, or for coprocessor registers, a string syntax that identifies the coprocessor and corresponds with how you intend to use the variable.

Registers available for use with named register variables on ARM architecture-based processors are shown in [Table 5-19](#).

Table 5-19 Named registers available on ARM architecture-based processors

Register	Character string for __asm	Processors
APSR	"apsr"	All processors
CPSR	"cpsr"	All processors
BASEPRI	"basepri"	Cortex-M3, Cortex-M4
BASEPRI_MAX	"basepri_max"	Cortex-M3, Cortex-M4
CONTROL	"control"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4
DSP	"dsp"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4
EAPSR	"eapsr"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4
EPSR	"epsr"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4
FAULTMASK	"faultmask"	Cortex-M3, Cortex-M4
IAPSR	"iapsr"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4
IEPSR	"iepsr"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4
IPSR	"ipsr"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4
MSP	"msp"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4

Table 5-19 Named registers available on ARM architecture-based processors (continued)

Register	Character string for <code>__asm</code>	Processors
PRIMASK	"primask"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4
PSP	"psp"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4
PSR	"psr"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4
r0 to r12	"r0" to "r12"	All processors
r13 or sp	"r13" or "sp"	All processors
r14 or lr	"r14" or "lr"	All processors
r15 or pc	"r15" or "pc"	All processors
SPSR	"spsr"	All processors, apart from Cortex-M series processors.
XPSR	"xpsr"	Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4

On targets with floating-point hardware, the registers of [Table 5-20](#) are also available for use with named register variables.

Table 5-20 Named registers available on targets with floating-point hardware

Register	Character string for <code>__asm</code>
FPSID	"fpsid"
FPSCR	"fpscr"
FPEXC	"fpexc"
FPIINST	"fpinst"
FPIINST2	"fpinst2"
FPSR	"fpsr"
MVFR0	"mvfr0"
MVFR1	"mvfr1"

———— **Note** —————

Some registers are not available on some architectures.

5.152.2 Usage

You can declare named register variables as global variables. You can declare some, but not all, named register variables as local variables. In general, do not declare VFP registers and core registers as local variables. Do not declare caller-save registers, such as R0, as local variables.

5.152.3 Examples

In [Example 5-12](#), `apSr` is declared as a named register variable for the "apSr" register:

Example 5-12 Named register variable for APSR

```
register unsigned int apSr __asm("apSr");
apSr = ~(~apSr | 0x40);
```

This generates the following instruction sequence:

```
MRS r0,APSR ; formerly CPSR
BIC r0,r0,#0x40
MSR CPSR_c, r0
```

In [Example 5-13](#), `PMCR` is declared as a register variable associated with coprocessor cp15, with `CRn = c9`, `CRm = c12`, `opcode1 = 0`, and `opcode2 = 0`, in an MCR or an MRC instruction:

Example 5-13 Named register variable for coprocessor register

```
register unsigned int PMCR __asm("cp15:0:c9:c12:0");

__inline void __reset_cycle_counter(void)
{
    PMCR = 4;
}
```

The disassembled output is as follows:

```
__reset_cycle_counter PROC
    MOV    r0,#4
    MCR   p15,#0x0,r0,c9,c12,#0
    BX    lr
    ENDP
```

In [Example 5-14](#), `cp15_control` is declared as a register variable that is used to access a coprocessor register. This example enables the MMU using CP15:

Example 5-14 Named register variable for coprocessor register to enable MMU

```
register unsigned int cp15_control __asm("cp15:0:c1:c0:0");
cp15_control |= 0x1;
```

The following instruction sequence is generated:

```
MRC   p15,#0x0,r0,c1,c0,#0
ORR   r0,r0,#1
MCR   p15,#0x0,r0,c1,c0,#0
```

[Example 5-15 on page 5-179](#) for Cortex-M3 declares `_msp`, `_control` and `_psp` as named register variables to set up stack pointers:

Example 5-15 Named register variables to set up stack pointers on Cortex-M3

```

register unsigned int _control __asm("control");
register unsigned int _msp     __asm("msp");
register unsigned int _psp     __asm("psp");void init(void)
{
    _msp = 0x30000000;        // set up Main Stack Pointer
    _control = _control | 3; // switch to User Mode with Process Stack
    _psp = 0x40000000;        // setup Process Stack Pointer
}

```

This generates the following instruction sequence:

```

init
MOV r0,#0x30000000
MSR MSP,r0
MRS r0,CONTROL
ORR r0,r0,#3
MSR CONTROL,r0
MOV r0,#0x40000000
MSR PSP,r0
BX lr

```

5.152.4 See also

- [Compiler support for accessing registers using named register variables on page 5-19](#) in *Using the Compiler*.

5.153 GNU built-in functions

These functions provide compatibility with GNU library header files. The functions are described in the GNU documentation. See <http://gcc.gnu.org>. See also `--gnu_version=version` on page 3-110.

5.153.1 Nonstandard functions

```
__builtin_alloca(), __builtin_bcmp(), __builtin_exit(), __builtin_gamma(),
__builtin_gammaf(), __builtin_gammal(), __builtin_index(),
__builtin_memcpy_chk(), __builtin_memmove_chk(), __builtin_memcpy(),
__builtin_memcpy_chk(), __builtin_memset_chk(), __builtin_object_size(),
__builtin_rindex(), __builtin_sprintf_chk(), __builtin_sprintf_chk(),
__builtin_stpcpy(), __builtin_stpcpy_chk(), __builtin_strcat_chk(),
__builtin_strcpy_chk(), __builtin_strcasecmp(), __builtin_strncasecmp(),
__builtin_strncat_chk(), __builtin_strncpy_chk(), __builtin_vsnprintf_chk(),
__builtin_vsprintf_chk().
```

5.153.2 C99 functions

```
__builtin_exit(), __builtin_acoshf(), __builtin_acoshl(), __builtin_acosh(),
__builtin_asinhf(), __builtin_asinhl(), __builtin_asinh(), __builtin_atanhf(),
__builtin_atanh(), __builtin_cabsf(), __builtin_cabs(),
__builtin_cabsf(), __builtin_cacosf(), __builtin_cacoshf(), __builtin_cacoshl(),
__builtin_cacosh(), __builtin_cacosl(), __builtin_cacos(), __builtin_cargf(),
__builtin_cargl(), __builtin_carg(), __builtin_casinf(), __builtin_casinhf(),
__builtin_casinh(), __builtin_casinl(), __builtin_casin(),
__builtin_catanf(), __builtin_catanhf(), __builtin_catanhl(),
__builtin_catanh(), __builtin_catanl(), __builtin_catan(), __builtin_cbrtf(),
__builtin_cbrtl(), __builtin_cbrt(), __builtin_ccosf(), __builtin_ccoshf(),
__builtin_ccoshl(), __builtin_ccosh(), __builtin_ccosl(), __builtin_ccos(),
__builtin_cexpf(), __builtin_cexpl(), __builtin_cexp(), __builtin_cimagf(),
__builtin_cimagl(), __builtin_cimag(), __builtin_clogf(), __builtin_clogl(),
__builtin_clog(), __builtin_conjf(), __builtin_conjl(), __builtin_conj(),
__builtin_copysignf(), __builtin_copysignl(), __builtin_copysign(),
__builtin_cpowf(), __builtin_cpowl(), __builtin_cpow(), __builtin_cprojf(),
__builtin_cprojl(), __builtin_cproj(), __builtin_creal(), __builtin_creal(),
__builtin_csinh(), __builtin_csinl(), __builtin_csin(), __builtin_csqrfl(),
__builtin_csqrfl(), __builtin_csqrfl(), __builtin_ctanf(), __builtin_ctanhf(),
__builtin_ctanh(), __builtin_ctanh(), __builtin_ctanh(), __builtin_ctanh(),
__builtin_erfcf(), __builtin_erfc(), __builtin_erfc(), __builtin_erff(),
__builtin_erfl(), __builtin_erf(), __builtin_exp2f(), __builtin_exp2l(),
__builtin_exp2(), __builtin_expm1f(), __builtin_expm1l(), __builtin_expm1(),
__builtin_fdimf(), __builtin_fdiml(), __builtin_fdim(), __builtin_fmaf(),
__builtin_fmal(), __builtin_fmaxf(), __builtin_fmaxl(), __builtin_fmax(),
__builtin_fma(), __builtin_fminf(), __builtin_fminl(), __builtin_fmin(),
__builtin_hypotf(), __builtin_hypotl(), __builtin_hypot(), __builtin_ilogbf(),
__builtin_ilogbl(), __builtin_ilogb(), __builtin_imaxabs(), __builtin_isblank(),
__builtin_isfinite(), __builtin_isinf(), __builtin_isnan(), __builtin_isnanf(),
__builtin_isnanl(), __builtin_isnormal(), __builtin_iswblank(),
__builtin_lgammaf(), __builtin_lgammal(), __builtin_lgamma(), __builtin_llabs(),
__builtin_llrintf(), __builtin_llrintl(), __builtin_llrint(),
__builtin_llroundf(), __builtin_llroundl(), __builtin_llround(),
__builtin_log1pf(), __builtin_log1pl(), __builtin_log1p(), __builtin_log2f(),
__builtin_log2l(), __builtin_log2(), __builtin_logbf(), __builtin_logbl(),
__builtin_logb(), __builtin_lrintf(), __builtin_lrintl(), __builtin_lrint(),
__builtin_lroundf(), __builtin_lroundl(), __builtin_lround(),
__builtin_nearbyintf(), __builtin_nearbyintl(), __builtin_nearbyint(),
__builtin_nextafterf(), __builtin_nextafterl(), __builtin_nextafter(),
__builtin_nexttowardf(), __builtin_nexttowardl(), __builtin_nexttoward(),
__builtin_remainderf(), __builtin_remainderl(), __builtin_remainder(),
```

```

__builtin_remquof(), __builtin_remquol(), __builtin_remquo(), __builtin_rintf(),
__builtin_rintl(), __builtin_rint(), __builtin_roundf(), __builtin_roundl(),
__builtin_round(), __builtin_scalblnf(), __builtin_scalblnl(),
__builtin_scalbln(), __builtin_scalbnf(), __builtin_calbnl(),
__builtin_scalbn(), __builtin_signbit(), __builtin_signbitf(),
__builtin_signbitl(), __builtin_sprintf(), __builtin_tgammaf(),
__builtin_tgamma(), __builtin_truncf(), __builtin_truncl(),
__builtin_trunc(), __builtin_vfscanf(), __builtin_vscanf(),
__builtin_vsnprintf(), __builtin_vsscanf().

```

5.153.3 C99 functions in the C90 reserved namespace

```

__builtin_acosf(), __builtin_acosl(), __builtin_asinf(), __builtin_asinl(),
__builtin_atan2f(), __builtin_atan2l(), __builtin_atanf(), __builtin_atanl(),
__builtin_ceilf(), __builtin_ceil(), __builtin_cosf(), __builtin_coshf(),
__builtin_coshl(), __builtin_cosl(), __builtin_expf(), __builtin_expl(),
__builtin_fabsf(), __builtin_fabsl(), __builtin_floorf(), __builtin_floorl(),
__builtin_fmodf(), __builtin_fmodl(), __builtin_frexp(), __builtin_frexp(),
__builtin_ldexpf(), __builtin_ldexpl(), __builtin_log10f(), __builtin_log10l(),
__builtin_logf(), __builtin_logl(), __builtin_modfl(), __builtin_modf(),
__builtin_powf(), __builtin_powl(), __builtin_sinf(), __builtin_sinhf(),
__builtin_sinhl(), __builtin_sinl(), __builtin_sqrtf(), sqrtl, __builtin_tanf(),
__builtin_tanhf(), __builtin_tanh(), __builtin_tanl().

```

5.153.4 C94 functions

```

__builtin_swalnum(), __builtin_iswalpha(), __builtin_iswcntrl(),
__builtin_iswdigit(), __builtin_iswgraph(), __builtin_iswlower(),
__builtin_iswprint(), __builtin_iswpunct(), __builtin_iswspace(),
__builtin_iswupper(), __builtin_iswxdigit(), __builtin_towlower(),
__builtin_towupper().

```

5.153.5 C90 functions

```

__builtin_abort(), __builtin_abs(), __builtin_acos(), __builtin_asin(),
__builtin_atan2(), __builtin_atan(), __builtin_calloc(), __builtin_ceil(),
__builtin_cosh(), __builtin_cos(), __builtin_exit(), __builtin_exp(),
__builtin_fabs(), __builtin_floor(), __builtin_fmod(), __builtin_fprintf(),
__builtin_fputc(), __builtin_fputs(), __builtin_frexp(), __builtin_fscanf(),
__builtin_isalnum(), __builtin_isalpha(), __builtin_iscntrl(),
__builtin_isdigit(), __builtin_isgraph(), __builtin_islower(),
__builtin_isprint(), __builtin_ispunct(), __builtin_isspace(),
__builtin_isupper(), __builtin_isxdigit(), __builtin_tolower(),
__builtin_toupper(), __builtin_labs(), __builtin_ldexp(), __builtin_log10(),
__builtin_log(), __builtin_malloc(), __builtin_memchr(), __builtin_memcmp(),
__builtin_memcpy(), __builtin_memset(), __builtin_modf(), __builtin_pow(),
__builtin_printf(), __builtin_putchar(), __builtin_puts(), __builtin_scanf(),
__builtin sinh(), __builtin sin(), __builtin_sprintf(), __builtin sprintf(),
__builtin_sqrt(), __builtin_sscanf(), __builtin_strcat(), __builtin strchr(),
__builtin_strcmp(), __builtin strcpy(), __builtin strcspn(),
__builtin strlen(), __builtin strncat(), __builtin strncmp(),
__builtin strncpy(), __builtin strpbrk(), __builtin strrchr(),
__builtin strspn(), __builtin strstr(), __builtin_tanh(), __builtin tan(),
__builtin_va_copy(), __builtin_va_end(), __builtin_va_start(),
__builtin_vfprintf(), __builtin_vprintf(), __builtin_vsprintf().

```

The `__builtin_va_list` type is also supported. It is equivalent to the `va_list` type declared in `stdarg.h`.

5.153.6 C99 floating-point functions

```
__builtin_huge_val(), __builtin_huge_valf(), __builtin_huge_vall(),
__builtin_inf(), __builtin_nan(), __builtin_nanf(), __builtin_nanl(),
__builtin_nans(), __builtin_nansf(), __builtin_nansl().
```

5.153.7 GNU atomic memory access functions

```
__sync_fetch_and_add(), __sync_fetch_and_sub(), __sync_fetch_and_or(),
__sync_fetch_and_and(), __sync_fetch_and_xor(), __sync_fetch_and_nand(),
__sync_add_and_fetch(), __sync_sub_and_fetch(), __sync_or_and_fetch(),
__sync_and_and_fetch(), __sync_xor_and_fetch(), __sync_nand_and_fetch(),
__sync_bool_compare_and_swap(), __sync_val_compare_and_swap(),
__sync_lock_test_and_set(), __sync_lock_release(), __sync_synchronize().
```

5.153.8 Other built-in functions

```
__builtin_clz(), __builtin_constant_p(), __builtin_ctz(), __builtin_ctzl(),
__builtin_ctzll(), __builtin_expect(), __builtin_ffs(), __builtin_ffsl(),
__builtin_ffsll(), __builtin_frame_address(), __builtin_prefetch(),
__builtin_return_address(), __builtin_popcount(), __builtin_signbit().
```

5.154 Predefined macros

This topic documents the predefined macros of the ARM compiler.

Table 5-21 lists the macro names predefined by the ARM compiler for C and C++. Where the value field is empty, the symbol is only defined.

Table 5-21 Predefined macros

Name	Value	When defined
<code>__arm__</code>	-	Always defined for the ARM compiler, even when you specify the <code>--thumb</code> option. See also <code>__ARMCC_VERSION</code> .
<code>__ARM_NEON__</code>	-	When compiler <code>--cpu</code> and <code>--fpu</code> options indicate that NEON is available. This macro can be used to conditionally include <code>arm_neon.h</code> , to permit the use of NEON intrinsics.
<code>__ARMCC_VERSION</code>	<i>ver</i>	Always defined. It is a decimal number, and is guaranteed to increase between releases. The format is <i>PVVbbbb</i> where: <ul style="list-style-type: none"> <i>P</i> is the major version <i>VV</i> is the minor version <i>bbbb</i> is the build number. <p style="text-align: center;">———— Note —————</p> Use this macro to distinguish between ARM Compiler 4.1 or later, and other tools that define <code>__arm__</code> .
<code>__APCS_INTERWORK</code>	-	When you specify the <code>--apcs /interwork</code> option or set the CPU architecture to ARMv5T or later.
<code>__APCS_ROPI</code>	-	When you specify the <code>--apcs /ropi</code> option.
<code>__APCS_RWPI</code>	-	When you specify the <code>--apcs /rwp</code> option.
<code>__APCS_FPIC</code>	-	When you specify the <code>--apcs /fpic</code> option.
<code>__ARRAY_OPERATORS</code>	-	In C++ compiler mode, to specify that array new and delete are enabled.
<code>__BASE_FILE__</code>	<i>name</i>	Always defined. Similar to <code>__FILE__</code> , but indicates the primary source file rather than the current one (that is, when the current file is an included file).
<code>__BIG_ENDIAN</code>	-	If compiling for a big-endian target.
<code>_BOOL</code>	-	In C++ compiler mode, to specify that <code>bool</code> is a keyword.
<code>__cplusplus</code>	-	In C++ compiler mode.
<code>__CC_ARM</code>	1	Always set to 1 for the ARM compiler, even when you specify the <code>--thumb</code> option.
<code>__CHAR_UNSIGNED__</code>	-	In GNU mode. It is defined if and only if <code>char</code> is an unsigned type.
<code>__DATE__</code>	<i>date</i>	Always defined.
<code>__EDG__</code>	-	Always defined.
<code>__EDG_IMPLICIT_USING_STD</code>	-	In C++ mode when you specify the <code>--using_std</code> option.

Table 5-21 Predefined macros (continued)

Name	Value	When defined
__EDG_VERSION__	-	Always set to an integer value that represents the version number of the <i>Edison Design Group</i> (EDG) front-end. For example, version 3.8 is represented as 308. <i>The version number of the EDG front-end does not necessarily match the version number of the ARM compiler toolchain.</i>
__EXCEPTIONS	1	In C++ mode when you specify the <code>--exceptions</code> option.
__FEATURE_SIGNED_CHAR	-	When you specify the <code>--signed_chars</code> option (used by <code>CHAR_MIN</code> and <code>CHAR_MAX</code>).
__FILE__	<i>name</i>	Always defined as a string literal.
__FP_FAST	-	When you specify the <code>--fpmode=fast</code> option.
__FP_FENV_EXCEPTIONS	-	When you specify the <code>--fpmode=ieee_full</code> or <code>--fpmode=ieee_fixed</code> options.
__FP_FENV_ROUNDING	-	When you specify the <code>--fpmode=ieee_full</code> option.
__FP_IEEE	-	When you specify the <code>--fpmode=ieee_full</code> , <code>--fpmode=ieee_fixed</code> , or <code>--fpmode=ieee_no_fenv</code> options.
__FP_INEXACT_EXCEPTION	-	When you specify the <code>--fpmode=ieee_full</code> option.
__GNUC__	<i>ver</i>	When you specify the <code>--gnu</code> option. It is an integer that shows the current major version of the GNU mode being used.
__GNUC_MINOR__	<i>ver</i>	When you specify the <code>--gnu</code> option. It is an integer that shows the current minor version of the GNU mode being used.
__GNUG__	<i>ver</i>	In GNU mode when you specify the <code>--cpp</code> option. It has the same value as <code>__GNUC__</code> .
__IMPLICIT_INCLUDE	-	When you specify the <code>--implicit_include</code> option.
__INTMAX_TYPE__	-	In GNU mode. It defines the correct underlying type for the <code>intmax_t</code> typedef.
__LINE__	<i>num</i>	Always set. It is the source line number of the line of code containing this macro.
__MODULE__	<i>mod</i>	Contains the filename part of the value of <code>__FILE__</code> .
__MULTIFILE	-	When you explicitly or implicitly use the <code>--multifile</code> option. ^a
__NO_INLINE__	-	When you specify the <code>--no_inline</code> option in GNU mode.
__OPTIMISE_LEVEL	<i>num</i>	Always set to 2 by default, unless you change the optimization level using the <code>-O<i>num</i></code> option. ^a
__OPTIMISE_SPACE	-	When you specify the <code>-Ospace</code> option.
__OPTIMISE_TIME	-	When you specify the <code>-Otime</code> option.
__OPTIMIZE__	-	When <code>-O1</code> , <code>-O2</code> , or <code>-O3</code> is specified in GNU mode.
__OPTIMIZE_SIZE__	-	When <code>-Ospace</code> is specified in GNU mode.

Table 5-21 Predefined macros (continued)

Name	Value	When defined
__PLACEMENT_DELETE	-	In C++ mode to specify that placement delete (that is, an operator delete corresponding to a placement operator new , to be called if the constructor throws an exception) is enabled. This is only relevant when using exceptions.
__PTRDIFF_TYPE__	-	In GNU mode. It defines the correct underlying type for the <code>ptrdiff_t</code> typedef.
__RTTI	-	In C++ mode when RTTI is enabled.
__sizeof_int	4	For <code>sizeof(int)</code> , but available in preprocessor expressions.
__sizeof_long	4	For <code>sizeof(long)</code> , but available in preprocessor expressions.
__sizeof_ptr	4	For <code>sizeof(void *)</code> , but available in preprocessor expressions.
__SIZE_TYPE__	-	In GNU mode. It defines the correct underlying type for the <code>size_t</code> typedef.
__SOFTFP__	-	If compiling to use the software floating-point calling standard and library. Set when you specify the <code>--fpu=softvfp</code> option for ARM or Thumb, or when you specify <code>--fpu=softvfp+vfpv2</code> for Thumb.
__STDC__	-	In all compiler modes.
__STDC_VERSION__	-	Standard version information.
__STRICT_ANSI__	-	When you specify the <code>--strict</code> option.
__SUPPORT_SNAN__	-	Support for signalling NaNs when you specify <code>--fpmode=ieee_fixed</code> or <code>--fpmode=ieee_full</code> .
__TARGET_ARCH_ARM	<i>num</i>	The number of the ARM base architecture of the target CPU irrespective of whether the compiler is compiling for ARM or Thumb. For possible values of <code>__TARGET_ARCH_ARM</code> in relation to the ARM architecture versions, see Table 5-22 on page 5-188 .
__TARGET_ARCH_THUMB	<i>num</i>	The number of the Thumb base architecture of the target CPU irrespective of whether the compiler is compiling for ARM or Thumb. The value is defined as zero if the target does not support Thumb. For possible values of <code>__TARGET_ARCH_THUMB</code> in relation to the ARM architecture versions, see Table 5-22 on page 5-188 .
__TARGET_ARCH_XX	-	<i>XX</i> represents the target architecture and its value depends on the target architecture. For example, if you specify the compiler options <code>--cpu=4T</code> or <code>--cpu=ARM7TDMI</code> then <code>__TARGET_ARCH_4T</code> is defined.
__TARGET_CPU_XX	-	<i>XX</i> represents the target CPU. The value of <i>XX</i> is derived from the <code>--cpu</code> compiler option, or the default if none is specified. For example, if you specify the compiler option <code>--cpu=ARM7TM</code> then <code>__TARGET_CPU_ARM7TM</code> is defined and no other symbol starting with <code>__TARGET_CPU_</code> is defined. If you specify the target architecture, then <code>__TARGET_CPU_generic</code> is defined. If the CPU name specified with <code>--cpu</code> is in lowercase, it is converted to uppercase. For example, <code>--cpu=Cortex-R4</code> results in <code>__TARGET_CPU_CORTEX_R4</code> being defined (rather than <code>__TARGET_CPU_Cortex_R4</code>). If the processor name contains hyphen (-) characters, these are mapped to an underscore (_). For example, <code>--cpu=ARM1136JF-S</code> is mapped to <code>__TARGET_CPU_ARM1136JF_S</code> .

Table 5-21 Predefined macros (continued)

Name	Value	When defined
__TARGET_FEATURE_DOUBLEWORD	-	ARMv5T and above.
__TARGET_FEATURE_DSPMUL	-	If the DSP-enhanced multiplier is available, for example ARMv5TE.
__TARGET_FEATURE_MULTIPLY	-	If the target architecture supports the long multiply instructions MULL and MULAL.
__TARGET_FEATURE_DIVIDE	-	If the target architecture supports the hardware divide instruction (that is, ARMv7-M or ARMv7-R).
__TARGET_FEATURE_MULTIPROCESSING	-	When you specify any of the following options: <ul style="list-style-type: none"> • --cpu=Cortex-A9 • --cpu=Cortex-A9.no_neon • --cpu=Cortex-A9.no_neon.no_vfp • --cpu=Cortex-A5 • --cpu=Cortex-A5.vfp • --cpu=Cortex-A5.neon • --cpu=Cortex-A15 • --cpu=Cortex-A15.no_neon • --cpu=Cortex-A15.no_neon.no_vfp • --cpu=Cortex-A7 • --cpu=Cortex-A7.no_neon • --cpu=Cortex-A7.no_neon.no_vfp
__TARGET_FEATURE_NEON	-	When you specify any of the following options: <ul style="list-style-type: none"> • --cpu=Cortex-A5.neon • --cpu=Cortex-A8 • --cpu=Cortex-A9 • --cpu=Cortex-A15 • --cpu=Cortex-A7 • --cpu=QSP
__TARGET_FEATURE_THUMB	-	If the target architecture supports Thumb, ARMv4T or later.

Table 5-21 Predefined macros (continued)

Name	Value	When defined
__TARGET_FPU_xx	-	<p>One of the following is set to indicate the FPU usage:</p> <ul style="list-style-type: none"> __TARGET_FPU_NONE __TARGET_FPU_VFP __TARGET_FPU_SOFTVFP <p>In addition, if compiling with one of the following <code>--fpu</code> options, the corresponding target name is set:</p> <ul style="list-style-type: none"> <code>--fpu=softvfp+vfpv2</code>, <code>__TARGET_FPU_SOFTVFP_VFPV2</code> <code>--fpu=softvfp+vfpv3</code>, <code>__TARGET_FPU_SOFTVFP_VFPV3</code> <code>--fpu=softvfp+vfpv3_fp16</code>, <code>__TARGET_FPU_SOFTVFP_VFPV3_FP16</code> <code>--fpu=softvfp+vfpv3_d16</code>, <code>__TARGET_FPU_SOFTVFP_VFPV3_D16</code> <code>--fpu=softvfp+vfpv3_d16_fp16</code>, <code>__TARGET_FPU_SOFTVFP_VFPV3_D16_FP16</code> <code>--fpu=vfpv2</code>, <code>__TARGET_FPU_VFPV2</code> <code>--fpu=vfpv3</code>, <code>__TARGET_FPU_VFPV3</code> <code>--fpu=vfpv3_fp16</code>, <code>__TARGET_FPU_VFPV3_FP16</code> <code>--fpu=vfpv3_d16</code>, <code>__TARGET_FPU_VFPV3_D16</code> <code>--fpu=vfpv3_d16_fp16</code>, <code>__TARGET_FPU_VFPV3_D16_FP16</code> <code>--fpu=vfpv4</code>, <code>__TARGET_FPU_VFPV4</code> <code>--fpu=vfpv4_d16</code>, <code>__TARGET_FPU_VFPV4_D16</code> <p>See <code>--fpu=name</code> on page 3-100 for more information.</p>
__TARGET_PROFILE_A		When you specify the <code>--cpu=7-A</code> option.
__TARGET_PROFILE_R		When you specify the <code>--cpu=7-R</code> option.
__TARGET_PROFILE_M		<p>When you specify any of the following options:</p> <ul style="list-style-type: none"> <code>--cpu=6-M</code> <code>--cpu=6S-M</code> <code>--cpu=7-M</code>
__thumb__	-	<p>When the compiler is in Thumb state. That is, you have either specified the <code>--thumb</code> option on the command-line or <code>#pragma thumb</code> in your source code.</p> <p style="text-align: center;">Note</p> <ul style="list-style-type: none"> The compiler might generate some ARM code even if it is compiling for Thumb. <code>__thumb</code> and <code>__thumb__</code> become defined or undefined when using <code>#pragma thumb</code> or <code>#pragma arm</code>, but do not change in cases where Thumb functions are generated as ARM code for other reasons (for example, a function specified as <code>__irq</code>).
__TIME__	<i>time</i>	Always defined.
__UINTMAX_TYPE__	-	In GNU mode. It defines the correct underlying type for the <code>uintmax_t</code> typedef.
__USER_LABEL_PREFIX__		In GNU mode. It defines an empty string. This macro is used by some of the Linux header files.
__VERSION__	<i>ver</i>	When you specify the <code>--gnu</code> option. It is a string that shows the current version of the GNU mode being used.
_WCHAR_T	-	In C++ mode, to specify that <code>wchar_t</code> is a keyword.

Table 5-21 Predefined macros (continued)

Name	Value	When defined
<code>__WCHAR_TYPE__</code>	-	In GNU mode. It defines the correct underlying type for the <code>wchar_t</code> typedef.
<code>__WCHAR_UNSIGNED__</code>	-	In GNU mode when you specify the <code>--cpp</code> option. It is defined if and only if <code>wchar_t</code> is an unsigned type.
<code>__WINT_TYPE__</code>	-	In GNU mode. It defines the correct underlying type for the <code>wint_t</code> typedef.

- a. ARM recommends that if you have source code reliant on the `__OPTIMISE_LEVEL` macro to determine whether or not `--multifile` is in effect, you change to using `__MULTIFILE`.

Table 5-22 shows the possible values for `__TARGET_ARCH_THUMB` (see Table 5-21 on page 5-183), and how these values relate to versions of the ARM architecture.

Table 5-22 Thumb architecture versions in relation to ARM architecture versions

ARM architecture	<code>__TARGET_ARCH_ARM</code>	<code>__TARGET_ARCH_THUMB</code>
v4	4	0
v4T	4	1
v5T, v5TE, v5TEJ	5	2
v6, v6K, v6Z	6	3
v6T2	6	4
v6-M, v6S-M	0	3
v7-A, v7-R	7	4
v7-M, v7E-M	0	4

5.154.1 See also

- [Built-in function name variables on page 5-189.](#)

5.155 Built-in function name variables

Table 5-23 lists built-in variables supported by the compiler for C and C++.

Table 5-23 built-in variables

Name	Value
__FUNCTION__	Holds the name of the function as it appears in the source. __FUNCTION__ is a constant string literal. You cannot use the preprocessor to join the contents to other text to form new tokens.
__PRETTY_FUNCTION__	Holds the name of the function as it appears pretty printed in a language-specific fashion. __PRETTY_FUNCTION__ is a constant string literal. You cannot use the preprocessor to join the contents to other text to form new tokens.

5.155.1 See also

- [Predefined macros on page 5-183.](#)

Chapter 6

C and C++ Implementation Details

This chapter describes the language implementation details for the compiler:

- *C and C++ implementation details* on page 6-2
 - *Character sets and identifiers* on page 6-3
 - *Basic data types* on page 6-5
 - *Operations on basic data types* on page 6-7
 - *Structures, unions, enumerations, and bitfields* on page 6-9
- *C++ implementation details* on page 6-14
 - *Using the ::operator new function* on page 6-15
 - *Tentative arrays* on page 6-16
 - *Old-style C parameters in C++ functions* on page 6-17
 - *Anachronisms* on page 6-18
 - *Template instantiation* on page 6-19
 - *Namespaces* on page 6-20
 - *C++ exception handling* on page 6-22
 - *Extern inline functions* on page 6-23.

6.1 C and C++ implementation details

The following language implementation details are common to both C and C++:

- *Character sets and identifiers* on page 6-3
- *Basic data types* on page 6-5
- *Operations on basic data types* on page 6-7
- *Structures, unions, enumerations, and bitfields* on page 6-9.

6.2 Character sets and identifiers

The following points apply to the character sets and identifiers expected by the compiler:

- Uppercase and lowercase characters are distinct in all internal and external identifiers. An identifier can also contain a dollar (\$) character unless the `--strict` compiler option is specified. To permit dollar signs in identifiers with the `--strict` option, also use the `--dollar` command-line option.
- Calling `setlocale(LC_CTYPE, "ISO8859-1")` makes the `isupper()` and `islower()` functions behave as expected over the full 8-bit Latin-1 alphabet, rather than over the 7-bit ASCII subset. The locale must be selected at link time.
- Source files are compiled according to the currently selected locale. You might have to select a different locale, with the `--locale` command-line option, if the source file contains non-ASCII characters. See [Compiler command-line options listed by group on page 3-4](#) in *Using the Compiler* for more information.
- The compiler supports multibyte character sets, such as Unicode.
- Other properties of the source character set are host-specific.

The properties of the execution character set are target-specific. The ARM C and C++ libraries support the ISO 8859-1 (Latin-1 Alphabet) character set with the following consequences:

- The execution character set is identical to the source character set.
- There are eight bits in a character in the execution character set.
- There are four characters (bytes) in an `int`. If the memory system is:
 - Little-endian** The bytes are ordered from least significant at the lowest address to most significant at the highest address.
 - Big-endian** The bytes are ordered from least significant at the highest address to most significant at the lowest address.
- In C all character constants have type `int`. In C++ a character constant containing one character has the type `char` and a character constant containing more than one character has the type `int`. Up to four characters of the constant are represented in the integer value. The last character in the constant occupies the lowest-order byte of the integer value. Up to three preceding characters are placed at higher-order bytes. Unused bytes are filled with the NULL (`\0`) character.
- [Table 6-1](#) lists all integer character constants, that contain a single character or character escape sequence, are represented in both the source and execution character sets.

Table 6-1 Character escape codes

Escape sequence	Char value	Description
<code>\a</code>	7	Attention (bell)
<code>\b</code>	8	Backspace
<code>\t</code>	9	Horizontal tab
<code>\n</code>	10	New line (line feed)
<code>\v</code>	11	Vertical tab
<code>\f</code>	12	Form feed

Table 6-1 Character escape codes (continued)

Escape sequence	Char value	Description
\r	13	Carriage return
\xnn	0xnn	ASCII code in hexadecimal
\nnn	0nnn	ASCII code in octal

- Characters of the source character set in string literals and character constants map identically into the execution character set.
- Data items of type **char** are unsigned by default. They can be explicitly declared as **signed char** or **unsigned char**:
 - the `--signed_chars` option can be used to make the **char** signed
 - the `--unsigned_chars` option can be used to make the **char** unsigned.

———— **Note** —————

Care must be taken when mixing translation units that have been compiled with and without the `--signed_chars` and `--unsigned_chars` options, and that share interfaces or data structures.

The ARM ABI defines **char** as an unsigned byte, and this is the interpretation used by the C++ libraries supplied with the ARM compilation tools.

- No locale is used to convert multibyte characters into the corresponding wide characters for a wide character constant. This is not relevant to the generic implementation.

6.3 Basic data types

The following basic data types are implemented in ARM C and C++:

6.3.1 Size and alignment of basic data types

Table 6-2 gives the size and natural alignment of the basic data types.

Table 6-2 Size and alignment of data types

Type	Size in bits	Natural alignment in bytes
<code>char</code>	8	1 (byte-aligned)
<code>short</code>	16	2 (halfword-aligned)
<code>int</code>	32	4 (word-aligned)
<code>long</code>	32	4 (word-aligned)
<code>long long</code>	64	8 (doubleword-aligned)
<code>float</code>	32	4 (word-aligned)
<code>double</code>	64	8 (doubleword-aligned)
<code>long double</code>	64	8 (doubleword-aligned)
All pointers	32	4 (word-aligned)
<code>bool</code> (C++ only)	8	1 (byte-aligned)
<code>_Bool</code> (C only ^a)	8	1 (byte-aligned)
<code>wchar_t</code> (C++ only)	16	2 (halfword-aligned)

a. `stdbool.h` can be used to define the `bool` macro in C.

Type alignment varies according to the context:

- Local variables are usually kept in registers, but when local variables spill onto the stack, they are always word-aligned. For example, a spilled local `char` variable has an alignment of 4.
- The natural alignment of a packed type is 1.

See [Structures, unions, enumerations, and bitfields on page 6-9](#) for more information.

6.3.2 Integer

Integers are represented in two's complement form. The low word of a `long long` is at the low address in little-endian mode, and at the high address in big-endian mode.

6.3.3 Float

Floating-point quantities are stored in IEEE format:

- `float` values are represented by IEEE single-precision values
- `double` and `long double` values are represented by IEEE double-precision values.

For **double** and **long double** quantities the word containing the sign, the exponent, and the most significant part of the mantissa is stored with the lower machine address in big-endian mode and at the higher address in little-endian mode. See [Operations on floating-point types on page 6-7](#) for more information.

6.3.4 Arrays and pointers

The following statements apply to all pointers to objects in C and C++, except pointers to members:

- Adjacent bytes have addresses that differ by one.
- The macro `NULL` expands to the value `0`.
- Casting between integers and pointers results in no change of representation.
- The compiler warns of casts between pointers to functions and pointers to data.
- The type `size_t` is defined as `unsigned int`.
- The type `ptrdiff_t` is defined as `signed int`.

6.4 Operations on basic data types

The ARM compiler performs the usual arithmetic conversions set out in relevant sections of the ISO C99 and ISO C++ standards. The following subtopics describe additional points that relate to arithmetic operations.

See also [Expression evaluation on page D-14](#).

6.4.1 Operations on integral types

The following statements apply to operations on the integral types:

- All signed integer arithmetic uses a two's complement representation.
- Bitwise operations on signed integral types follow the rules that arise naturally from two's complement representation. No sign extension takes place.
- Right shifts on signed quantities are arithmetic.
- For values of type `int`,
 - Shifts outside the range 0 to 127 are undefined.
 - Left shifts of more than 31 give a result of zero.
 - Right shifts of more than 31 give a result of zero from a shift of an unsigned value or positive signed value. They yield `-1` from a shift of a negative signed value.
- For values of type `long long`, shifts outside the range 0 to 63 are undefined.
- The remainder on integer division has the same sign as the numerator, as mandated by the ISO C99 standard.
- If a value of integral type is truncated to a shorter signed integral type, the result is obtained by discarding an appropriate number of most significant bits. If the original number is too large, positive or negative, for the new type, there is no guarantee that the sign of the result is going to be the same as the original.
- A conversion between integral types does not raise an exception.
- Integer overflow does not raise an exception.
- Integer division by zero returns zero by default.

6.4.2 Operations on floating-point types

The following statements apply to operations on floating-point types:

- Normal IEEE 754 rules apply.
- Rounding is to the nearest representable value by default.
- Floating-point exceptions are disabled by default.

Also, see [--fpmode=model on page 3-97](#).

———— **Note** —————

The IEEE 754 standard for floating-point processing states that the default action to an exception is to proceed without a trap. You can modify floating-point error handling by tailoring the functions and definitions in `fenv.h`. See [Modification of C library functions for error signaling, error handling, and program exit on page 2-80](#) in *Using ARM C and C++ Libraries and Floating-Point Support* for more information.

6.4.3 Pointer subtraction

The following statements apply to all pointers in C. They also apply to pointers in C++, other than pointers to members:

- When one pointer is subtracted from another, the difference is the result of the expression:
 $((\text{int})a - (\text{int})b) / (\text{int})\text{sizeof}(\text{type pointed to})$
- If the pointers point to objects whose alignment is the same as their size, this alignment ensures that division is exact.
- If the pointers point to objects whose alignment is less than their size, such as packed types and most **structs**, both pointers must point to elements of the same array.

6.5 Structures, unions, enumerations, and bitfields

This topic describes the implementation of the structured data types union, enum, and struct. It also discusses structure padding and bitfield implementation.

See [Anonymous classes, structures and unions on page 4-38](#) for more information.

6.5.1 Unions

When a member of a **union** is accessed using a member of a different type, the resulting value can be predicted from the representation of the original type. No error is given.

6.5.2 Enumerations

An object of type **enum** is implemented in the smallest integral type that contains the range of the **enum**.

In C mode, and in C++ mode without `--enum_is_int`, if an **enum** contains only positive enumerator values, the storage type of the **enum** is the first *unsigned* type from the following list, according to the range of the enumerators in the **enum**. In other modes, and in cases where an **enum** contains any negative enumerator values, the storage type of the **enum** is the first of the following, according to the range of the enumerators in the **enum**:

- **unsigned char** if not using `--enum_is_int`
- **signed char** if not using `--enum_is_int`
- **unsigned short** if not using `--enum_is_int`
- **signed short** if not using `--enum_is_int`
- **signed int**
- **unsigned int** except C with `--strict`
- **signed long long** except C with `--strict`
- **unsigned long long** except C with `--strict`.

————— **Note** —————

- In RVCT 4.0, the storage type of the **enum** being the first unsigned type from the list is only applicable in GNU (`--gnu`) mode.
 - In ARM Compiler 4.1 and later, the storage type of the **enum** being the first unsigned type from the list applies irrespective of mode.
-

Implementing **enum** in this way can reduce data size. The command-line option `--enum_is_int` forces the underlying type of **enum** to at least as wide as **int**.

See the description of C language mappings in the *Procedure Call Standard for the ARM Architecture* specification for more information.

————— **Note** —————

Care must be taken when mixing translation units that have been compiled with and without the `--enum_is_int` option, and that share interfaces or data structures.

Handling values that are out of range

In strict C, enumerator values must be representable as **ints**, for example, they must be in the range -2147483648 to +2147483647, inclusive. In some earlier releases of RVCT out-of-range values were cast to **int** without a warning (unless you specified the `--strict` option).

In RVCT v2.2 and later, a Warning is issued for out-of-range enumerator values:

```
#66: enumeration value is out of "int" range
```

Such values are treated the same way as in C++, that is, they are treated as **unsigned int**, **long long**, or **unsigned long long**.

To ensure that out-of-range Warnings are reported, use the following command to change them into Errors:

```
armcc --diag_error=66 ...
```

6.5.3 Structures

The following points apply to:

- all C structures
- all C++ structures and classes not using virtual functions or base classes.

Structure alignment

The alignment of a nonpacked structure is the maximum alignment required by any of its fields.

Field alignment

Structures are arranged with the first-named component at the lowest address. Fields are aligned as follows:

- A field with a **char** type is aligned to the next available byte.
- A field with a **short** type is aligned to the next even-addressed byte.
- In RVCT v2.0 and above, **double** and **long long** data types are eight-byte aligned. This enables efficient use of the LDRD and STRD instructions in ARMv5TE and above.
- Bitfield alignment depends on how the bitfield is declared. See [Bitfields in packed structures on page 6-13](#) for more information.
- All other types are aligned on word boundaries.

Structures can contain padding to ensure that fields are correctly aligned and that the structure itself is correctly aligned. [Figure 6-1](#) shows an example of a conventional, nonpacked structure. Bytes 1, 2, and 3 are padded to ensure correct field alignment. Bytes 11 and 12 are padded to ensure correct structure alignment. The `sizeof()` function returns the size of the structure including padding.

```
struct {char c; int x; short s} ex1;
```

0	1	2	3
c	padding		
4	5	7	8
x			
9	10	11	12
s		padding	

Figure 6-1 Conventional nonpacked structure example

The compiler pads structures in one of the following ways, according to how the structure is defined:

- Structures that are defined as **static** or **extern** are padded with zeros.

- Structures on the stack or heap, such as those defined with `malloc()` or `auto`, are padded with whatever is previously stored in those memory locations. You cannot use `memcmp()` to compare padded structures defined in this way (see [Figure 6-1 on page 6-10](#)).

Use the `--remarks` option to view the messages that are generated when the compiler inserts padding in a `struct`.

Structures with empty initializers are permitted in C++:

```
struct
{
    int x;
} X = { };
```

However, if you are compiling C, or compiling C++ with the `--cpp` and `--c90` options, an error is generated.

6.5.4 Packed structures

A packed structure is one where the alignment of the structure, and of the fields within it, is always 1.

You can pack specific structures with the `__packed` qualifier. Alternatively, you can use `#pragma pack(n)` to make sure that any structures with unaligned data are packed. There is no command-line option to change the default packing of structures.

6.5.5 Bitfields

In nonpacked structures, the ARM compiler allocates bitfields in *containers*. A container is a correctly aligned object of a declared type.

Bitfields are allocated so that the first field specified occupies the lowest-addressed bits of the word, depending on configuration:

Little-endian Lowest addressed means least significant.

Big-endian Lowest addressed means most significant.

A bitfield container can be any of the integral types.

———— **Note** —————

In strict 1990 ISO Standard C, the only types permitted for a bit field are `int`, `signed int`, and `unsigned int`. For non-`int` bitfields, the compiler displays an error.

A plain bitfield, declared without either `signed` or `unsigned` qualifiers, is treated as `unsigned`. For example, `int x:10` allocates an unsigned integer of 10 bits.

A bitfield is allocated to the first container of the correct type that has a sufficient number of unallocated bits, for example:

```
struct X
{
    int x:10;
    int y:20;
};
```

The first declaration creates an integer container and allocates 10 bits to `x`. At the second declaration, the compiler finds the existing integer container with a sufficient number of unallocated bits, and allocates `y` in the same container as `x`.

A bitfield is wholly contained within its container. A bitfield that does not fit in a container is placed in the next container of the same type. For example, the declaration of `z` overflows the container if an additional bitfield is declared for the structure:

```
struct X
{
    int x:10;
    int y:20;
    int z:5;
};
```

The compiler pads the remaining two bits for the first container and assigns a new integer container for `z`.

Bitfield containers can *overlap* each other, for example:

```
struct X
{
    int x:10;
    char y:2;
};
```

The first declaration creates an integer container and allocates 10 bits to `x`. These 10 bits occupy the first byte and two bits of the second byte of the integer container. At the second declaration, the compiler checks for a container of type `char`. There is no suitable container, so the compiler allocates a new correctly aligned `char` container.

Because the natural alignment of `char` is 1, the compiler searches for the first byte that contains a sufficient number of unallocated bits to completely contain the bitfield. In the example structure, the second byte of the `int` container has two bits allocated to `x`, and six bits unallocated. The compiler allocates a `char` container starting at the second byte of the previous `int` container, skips the first two bits that are allocated to `x`, and allocates two bits to `y`.

If `y` is declared `char y:8`, the compiler pads the second byte and allocates a new `char` container to the third byte, because the bitfield cannot overflow its container. Figure 6-2 shows the bitfield allocation for the following example structure:

```
struct X
{
    int x:10;
    char y:8;
};
```

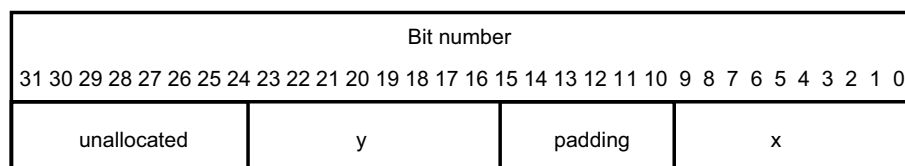


Figure 6-2 Bitfield allocation 1

———— **Note** ————

The same basic rules apply to bitfield declarations with different container types. For example, adding an `int` bitfield to the example structure gives:

```
struct X
{
    int x:10;
```

```

char y:8;
int z:5;
}

```

The compiler allocates an `int` container starting at the same location as the `int x:10` container and allocates a byte-aligned `char` and 5-bit bitfield, see [Figure 6-3](#).

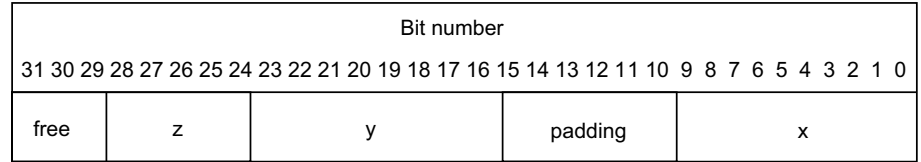


Figure 6-3 Bitfield allocation 2

You can explicitly pad a bitfield container by declaring an unnamed bitfield of size zero. A bitfield of zero size fills the container up to the end if the container is not empty. A subsequent bitfield declaration starts a new empty container.

6.5.6 Bitfields in packed structures

Bitfield containers in packed structures have an alignment of 1. Therefore, the maximum bit padding for a bitfield in a packed structure is 7 bits. For an unpacked structure, the maximum padding is $8 * \text{sizeof}(\text{container-type}) - 1$ bits.

6.6 C++ implementation details

The following language implementation details are specific to C++:

- *Using the ::operator new function* on page 6-15
- *Tentative arrays* on page 6-16
- *Old-style C parameters in C++ functions* on page 6-17
- *Anachronisms* on page 6-18
- *Template instantiation* on page 6-19
- *Namespaces* on page 6-20
- *C++ exception handling* on page 6-22
- *Extern inline functions* on page 6-23.

6.7 Using the `::operator new` function

In accordance with the ISO C++ Standard, the `::operator new(std::size_t)` throws an exception when memory allocation fails rather than raising a signal. If the exception is not caught, `std::terminate()` is called.

The compiler option `--force_new_nothrow` turns all `new` calls in a compilation into calls to `::operator new(std::size_t, std::nothrow_t&)` or `operator new[](std::size_t, std::nothrow_t&)`. However, this does not affect `operator new` calls in libraries, nor calls to any class-specific `operator new`. See `--force_new_nothrow`, `--no_force_new_nothrow` on page 3-94 for more information.

6.7.1 Legacy support

In RVCT v2.0, when the `::operator new` function ran out of memory, it raised the signal **SIGOUTOFHEAP**, instead of throwing a C++ exception. See *ISO-compliant implementation of signals supported by the `signal()` function in the C library and additional type arguments on page 2-110* in *Using ARM C and C++ Libraries and Floating-Point Support*.

In the current release, it is possible to install a `new_handler` to raise a signal and so restore the RVCT v2.0 behavior.

———— **Note** —————

Do not rely on the implementation details of this behavior, because it might change in future releases.

6.8 Tentative arrays

The ADS v1.2 and RVCT v1.2 C++ compilers enabled you to use tentative, that is, incomplete array declarations, for example, `int a[]`. You cannot use tentative arrays when compiling C++ with the RVCT v2.x compilers or later, or with ARM Compiler 4.1 or later.

6.9 Old-style C parameters in C++ functions

The ADS v1.2 and RVCT v1.2 C++ compilers enabled you to use old-style C parameters in C++ functions. That is,

```
void f(x) int x; { }
```

In the RVCT v2.x compilers or above, you must use the `--anachronisms` compiler option if your code contains any old-style parameters in functions. The compiler warns you if it finds any instances.

6.10 Anachronisms

The following anachronisms are accepted when you enable anachronisms using the `--anachronisms` option:

- **overload** is permitted 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, because these must always be defined.
- The number of elements in an array can be specified in an array delete operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name can be omitted in a base class initializer if there is only one immediate base class.
- Assignment to the `this` pointer in constructors and destructors is permitted.
- A bound function pointer, that is, a pointer to a member function for a given object, can be cast to a pointer to a function.
- A nested class name can be used as a non-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 can 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 can be initialized from an rvalue of the class type or a class derived from that class type. No, additional, temporary is used.
- A function with old-style parameter declarations is permitted and can participate in function overloading as if 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 declares the overloading of two functions named `f`:

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

———— **Note** —————

In C, this code is legal but has a different meaning. A tentative declaration of `f` is followed by its definition.

6.11 Template instantiation

The compiler does all template instantiations automatically, and makes sure there is only one definition of each template entity left after linking. The compiler does this by emitting template entities in named common sections. Therefore, all duplicate common sections, that is, common sections with the same name, are eliminated by the linker.

———— **Note** —————

You can limit the number of concurrent instantiations of a given template with the `--pending_instantiations` compiler option.

See also [--pending_instantiations=n on page 3-169](#) for more information.

6.11.1 Implicit inclusion

When implicit inclusion is enabled, the compiler assumes that if it requires a definition to instantiate a template entity declared in a `.h` file it can implicitly include the corresponding `.cc` file to get the source code for the definition. For example, if a template entity `ABC::f` is declared in file `xyz.h`, and an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears in the source code processed by the compilation, then the compiler checks to see if a file `xyz.cc` exists. If this file exists, the compiler processes the file as if it were included at the end of the main source file.

To find the template definition file for a given template entity the compiler has to know the full path name of the file where the template is declared and whether the file is included using the system include syntax, for example, `#include <file.h>`. This information is not available for preprocessed source containing `#line` directives. Consequently, the compiler does not attempt implicit inclusion for source code containing `#line` directives.

The compiler looks for the definition-file suffixes `.cc` and `.CC`.

You can turn implicit inclusion mode on or off with the command-line options `--implicit_include` and `--no_implicit_include`.

Implicit inclusions are only performed during the normal compilation of a file, that is, when not using the `-E` command-line option.

See [Chapter 3 Compiler Command-line Options](#) for more information.

6.12 Namespaces

When doing name lookup in a template instantiation, some names must be found in the context of the template definition. Other names can be found in the context of the template instantiation. The compiler implements two different instantiation lookup algorithms:

- the algorithm mandated by the standard, and referred to as dependent name lookup.
- the algorithm that exists before dependent name lookup is implemented.

Dependent name lookup is done in strict mode, unless explicitly disabled by another command-line option, or when dependent name processing is enabled by either a configuration flag or a command-line option.

6.12.1 Dependent name lookup processing

When doing dependent name lookup, the compiler implements the instantiation name lookup rules specified in the standard. This processing requires that nonclass prototype instantiations be done. This in turn requires that the code be written using the typename and template keywords as required by the standard.

6.12.2 Lookup using the referencing context

When not using dependent name lookup, the compiler uses a name lookup algorithm that approximates the two-phase lookup rule of the standard, but in a way that is more compatible with existing code and existing compilers.

When a name is looked up as part of a template instantiation, but is not found in the local context of the instantiation, it is looked up in a synthesized instantiation context. This synthesized instantiation context includes both names from the context of the template definition and names from the context of the instantiation. For example:

```
namespace N
{
    int g(int);
    int x = 0;
    template <class T> struct A
    {
        T f(T t) { return g(t); }
        T f() { return x; }
    };
}
namespace M {
    int x = 99;
    double g(double);
    N::A<int> ai;
    int i = ai.f(0);           // N::A<int>::f(int) calls N::g(int)
    int i2 = ai.f();          // N::A<int>::f() returns 0 (= N::x)
    N::A<double> ad;
    double d = ad.f(0);       // N::A<double>::f(double) calls M::g(double)
    double d2 = ad.f();       // N::A<double>::f() also returns 0 (= N::x)
}
```

The lookup of names in template instantiations does not conform to the rules in the standard in the following respects:

- Although only names from the template definition context are considered for names that are not functions, the lookup is not limited to those names visible at the point where the template is defined.

- Functions from the context where the template is referenced are considered for all function calls in the template. Functions from the referencing context are only visible for dependent function calls.

6.12.3 Argument-dependent lookup

When argument-dependent lookup is enabled, functions that are made visible using argument-dependent lookup can overload with those made visible by normal lookup. The standard requires that this overloading occur even when the name found by normal lookup is a block extern declaration. The compiler does this overloading, but in default mode, argument-dependent lookup is suppressed when the normal lookup finds a block extern.

This means a program can have different behavior, depending on whether it is compiled with or without argument-dependent lookup, even if the program makes no use of namespaces. For example:

```
struct A { };
A operator+(A, double);
void f()
{
    A a1;
    A operator+(A, int);
    a1 + 1.0;           // calls operator+(A, double) with arg-dependent lookup
}                      // enabled but otherwise calls operator+(A, int);
```

6.13 C++ exception handling

The ARM compilation tools fully support C++ exception handling. However, the compiler does not support this by default. You must enable C++ exception handling with the `--exceptions` option. See [--exceptions, --no_exceptions on page 3-87](#) for more information.

Note

The Rogue Wave Standard C++ Library is provided with C++ exceptions enabled.

You can exercise limited control over exception table generation.

6.13.1 Function unwinding at runtime

By default, functions compiled with `--exceptions` can be unwound at runtime. See [--exceptions, --no_exceptions on page 3-87](#) for more information. *Function unwinding* includes destroying C++ automatic variables, and restoring register values saved in the stack frame. Function unwinding is implemented by emitting an exception table describing the operations to be performed.

You can enable or disable unwinding for specific functions with the pragmas `#pragma exceptions_unwind` and `#pragma no_exceptions_unwind`, see [Pragmas on page 5-85](#) for more information. The `--exceptions_unwind` option sets the initial value of this pragma.

Disabling function unwinding for a function has the following effects:

- Exceptions cannot be thrown through that function at runtime, and no stack unwinding occurs for that throw. If the throwing language is C++, then `std::terminate` is called.
- A very compact exception table representation can be used to describe the function, that assists smart linkers with table optimization.
- Function inlining is restricted, because the caller and callee must interact correctly.

Therefore, `#pragma no_exceptions_unwind` can be used to forcibly prevent unwinding in a way that requires no additional source decoration.

By contrast, in C++ an empty function exception specification permits unwinding as far as the protected function, then calls `std::unexpected()` in accordance with the ISO C++ Standard.

6.14 Extern inline functions

The ISO C++ Standard requires inline functions to be defined wherever you use them. To prevent the clashing of multiple out-of-line copies of inline functions, the C++ compiler emits out-of-line extern functions in common sections.

6.14.1 Out-of-line inline functions

The compiler emits inline functions out-of-line, in the following cases:

- The address of the function is taken, for example:

```
inline int g()
{
    return 1;
}
int (*fp)() = &g;
```

- The function cannot be inlined, for example, a recursive function:

```
inline unsigned int fact(unsigned int n) {
    return n < 2 ? 1 : n * fact(n - 1);
}
```

- The heuristic used by the compiler decides that it is better not to inline the function. This heuristic is influenced by `-Ospace` and `-Otime`. If you use `-Otime`, the compiler inlines more functions. You can override this heuristic by declaring a function with `__forceinline`. For example:

```
__forceinline int g()
{
    return 1;
}
```

See also [--forceinline](#) on page 3-95 for more information.

Appendix A

ARMv6 SIMD Instruction Intrinsics

This appendix describes the ARMv6 SIMD instruction intrinsics:

- [ARMv6 SIMD intrinsics by prefix](#) on page A-3
- [ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags](#) on page A-5
- [ARMv6 SIMD intrinsics, compatible processors and architectures](#) on page A-9
- [ARMv6 SIMD instruction intrinsics and APSR GE flags](#) on page A-10
- [__qadd16 intrinsic](#) on page A-11
- [__qadd8 intrinsic](#) on page A-12
- [__qasx intrinsic](#) on page A-13
- [__qsax intrinsic](#) on page A-14
- [__qsub16 intrinsic](#) on page A-15
- [__qsub8 intrinsic](#) on page A-16
- [__sadd16 intrinsic](#) on page A-17
- [__sadd8 intrinsic](#) on page A-18
- [__sasx intrinsic](#) on page A-19
- [__sel intrinsic](#) on page A-20
- [__shadd16 intrinsic](#) on page A-21
- [__shadd8 intrinsic](#) on page A-22
- [__shasx intrinsic](#) on page A-23
- [__shsax intrinsic](#) on page A-24
- [__shsub16 intrinsic](#) on page A-25
- [__shsub8 intrinsic](#) on page A-26
- [__smlad intrinsic](#) on page A-27
- [__smladx intrinsic](#) on page A-28

- [__smlald intrinsic](#) on page A-29
- [__smlaldx intrinsic](#) on page A-30
- [__smlsd intrinsic](#) on page A-31
- [__smlsdx intrinsic](#) on page A-32
- [__smlsld intrinsic](#) on page A-33
- [__smlsldx intrinsic](#) on page A-34
- [__smuad intrinsic](#) on page A-35
- [__smuadx intrinsic](#) on page A-36
- [__smusd intrinsic](#) on page A-37
- [__smusdx intrinsic](#) on page A-38
- [__ssat16 intrinsic](#) on page A-39
- [__ssax intrinsic](#) on page A-40
- [__ssub16 intrinsic](#) on page A-41
- [__ssub8 intrinsic](#) on page A-42
- [__sxtab16 intrinsic](#) on page A-43
- [__sxtb16 intrinsic](#) on page A-44
- [__uadd16 intrinsic](#) on page A-45
- [__uadd8 intrinsic](#) on page A-46
- [__uax intrinsic](#) on page A-47
- [__uhadd16 intrinsic](#) on page A-48
- [__uhadd8 intrinsic](#) on page A-49
- [__uhasx intrinsic](#) on page A-50
- [__uhsax intrinsic](#) on page A-51
- [__uhsub16 intrinsic](#) on page A-52
- [__uhsub8 intrinsic](#) on page A-53
- [__uqadd16 intrinsic](#) on page A-54
- [__uqadd8 intrinsic](#) on page A-55
- [__uqasx intrinsic](#) on page A-56
- [__uqsax intrinsic](#) on page A-57
- [__uqsub16 intrinsic](#) on page A-58
- [__uqsub8 intrinsic](#) on page A-59
- [__usad8 intrinsic](#) on page A-60
- [__usada8 intrinsic](#) on page A-61
- [__usax intrinsic](#) on page A-62
- [__usat16 intrinsic](#) on page A-63
- [__usub16 intrinsic](#) on page A-64
- [__usub8 intrinsic](#) on page A-65
- [__uxtab16 intrinsic](#) on page A-66
- [__uxtb16 intrinsic](#) on page A-67.

A.1 ARMv6 SIMD intrinsics by prefix

Table A-1 shows the intrinsics according to prefix name.

The `__sel()` intrinsic falls outside the classifications shown in the table. This intrinsic selects bytes according to GE bit values.

Table A-1 ARMv6 SIMD intrinsics by prefix

ARMv6 SIMD instruction intrinsics grouped by prefix						
Intrinsic classification	<code>__sa</code>	<code>__qb</code>	<code>__shc</code>	<code>__ud</code>	<code>__uqe</code>	<code>__uhf</code>
Byte addition	<code>__sadd8</code>	<code>__qadd8</code>	<code>__shadd8</code>	<code>__uadd8</code>	<code>__uqadd8</code>	<code>__uhadd8</code>
Byte subtraction	<code>__ssub8</code>	<code>__qsub8</code>	<code>__shsub8</code>	<code>__usub8</code>	<code>__uqsub8</code>	<code>__uhsub8</code>
Halfword addition	<code>__sadd16</code>	<code>__qadd16</code>	<code>__shadd16</code>	<code>__uadd16</code>	<code>__uqadd16</code>	<code>__uhadd16</code>
Halfword subtraction	<code>__ssub16</code>	<code>__qsub16</code>	<code>__shsub16</code>	<code>__usub16</code>	<code>__uqsub16</code>	<code>__uhsub16</code>
Exchange halfwords within one operand, add high halfwords, subtract low halfwords	<code>__sasx</code>	<code>__qasx</code>	<code>__shasx</code>	<code>__uasx</code>	<code>__uqasx</code>	<code>__uhasx</code>
Exchange halfwords within one operand, subtract high halfwords, add low halfwords	<code>__ssax</code>	<code>__qsax</code>	<code>__shsax</code>	<code>__usax</code>	<code>__uqsax</code>	<code>__uhsax</code>
Unsigned sum of absolute difference	-	-	-	<code>__usad8</code>	-	-
Unsigned sum of absolute difference and accumulate	-	-	-	<code>__usada8</code>	-	-
Saturation to selected width	<code>__ssat16</code>	-	-	<code>__usat16</code>	-	-
Extract values (bit positions [23:16][7:0]), zero-extend to 16 bits	-	-	-	<code>__uxtb16</code>	-	-
Extract values (bit positions [23:16][7:0]) from second operand, zero-extend to 16 bits, add to first operand	-	-	-	<code>__uxtab16</code>	-	-
Sign-extend	<code>__sxtb16</code>	-	-	-	-	-
Sign-extend, add	<code>__sxtab16</code>	-	-	-	-	-
Signed multiply, add products	<code>__smuad</code>	-	-	-	-	-
Exchange halfwords of one operand, signed multiply, add products	<code>__smuadx</code>	-	-	-	-	-
Signed multiply, subtract products	<code>__smusd</code>	-	-	-	-	-
Exchange halfwords of one operand, signed multiply, subtract products	<code>__smusdx</code>	-	-	-	-	-
Signed multiply, add both results to another operand	<code>__sm1ad</code>	-	-	-	-	-
Exchange halfwords of one operand, perform 2x16-bit multiplication, add both results to another operand	<code>__sm1adx</code>	-	-	-	-	-
Perform 2x16-bit multiplication, add both results to another operand	<code>__sm1ald</code>	-	-	-	-	-

Table A-1 ARMv6 SIMD intrinsics by prefix (continued)

ARMv6 SIMD instruction intrinsics grouped by prefix						
Intrinsic classification	__sa ^a	__qb ^b	__sh ^c	__ud ^d	__uqe ^e	__uh ^f
Exchange halfwords of one operand, perform 2x16-bit multiplication, add both results to another operand	__sm1a1dx	-	-	-	-	-
Perform 2x16-bit signed multiplications, take difference of products, subtracting high halfword product from low halfword product, and add difference to a 32-bit accumulate operand	__sm1sd	-	-	-	-	-
Exchange halfwords of one operand, perform two signed 16-bit multiplications, add difference of products to a 32-bit accumulate operand	__sm1sdx	-	-	-	-	-
Perform 2x16-bit signed multiplications, take difference of products, subtracting high halfword product from low halfword product, add difference to a 64-bit accumulate operand	__sm1s1d	-	-	-	-	-
Exchange halfwords of one operand, perform 2x16-bit multiplications, add difference of products to a 64-bit accumulate operand	__sm1s1dx	-	-	-	-	-

- a. Signed
- b. Signed saturating
- c. Signed halving
- d. Unsigned
- e. Unsigned saturating
- f. Unsigned halving.

A.2 ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags

Table A-2 ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags

Intrinsic	Summary description	Byte lanes		Affected flags
		Returns	Operands	
__qadd16	2 x 16-bit addition, saturated to range $-2^{15} \leq x \leq 2^{15} - 1$.	int16x2	int16x2, int16x2	None
__qadd8	4 x 8-bit addition, saturated to range $-2^7 \leq x \leq 2^7 - 1$.	int8x4	int8x4, int8x4	None
__qasx	Exchange halfwords of second operand, add high halfwords, subtract low halfwords, saturating in each case.	int16x2	int16x2, int16x2	None
__qsax	Exchange halfwords of second operand, subtract high halfwords, add low halfwords, saturating in each case.	int16x2	int16x2, int16x2	None
__qsub16	2 x 16-bit subtraction with saturation.	int16x2	int16x2, int16x2	None
__qsub8	4 x 8-bit subtraction with saturation.	int8x4	int8x4, int8x4	None
__sadd16	2 x 16-bit signed addition.	int16x2	int16x2, int16x2	APSR.GE bits
__sadd8	4 x 8-bit signed addition.	int8x4	int8x4, int8x4	APSR.GE bits
__sasx	Exchange halfwords of second operand, add high halfwords, subtract low halfwords.	int16x2	int16x2, int16x2	APSR.GE bits
__se1	Select each byte of the result from either the first operand or the second operand, according to the values of the GE bits. For each result byte, if the corresponding GE bit is set, the byte from the first operand is selected, otherwise the byte from the second operand is selected. Because of the way that int16x2 operations set two (duplicate) GE bits per value, the __se1 intrinsic works equally well on (u)int16x2 and (u)int8x4 data.	uint8x4	uint8x4, uint8x4	None
__shadd16	2x16-bit signed addition, halving the results.	int16x2	int16x2, int16x2	None
__shadd8	4x8-bit signed addition, halving the results.	int8x4	int8x4, int8x4	None
__shasx	Exchange halfwords of the second operand, add high halfwords and subtract low halfwords, halving the results.	int16x2	int16x2, int16x2	None
__shsax	Exchange halfwords of the second operand, subtract high halfwords and add low halfwords, halving the results.	int16x2	int16x2, int16x2	None
__shsub16	2x16-bit signed subtraction, halving the results.	int16x2	int16x2, int16x2	None
__shsub8	4x8-bit signed subtraction, halving the results.	int8x4	int8x4, int8x4	None

Table A-2 ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags (continued)

Intrinsic	Summary description	Byte lanes		Affected flags
		Returns	Operands	
__sm1ad	2x16-bit multiplication, adding both results to third operand.	int32	int16x2, int16x2, int32	Q bit
__sm1adx	Exchange halfwords of the second operand, 2x16-bit multiplication, adding both results to third operand.	int16x2	int16x2, int16x2	Q bit
__sm1ald	2x16-bit multiplication, adding both results to third operand. Overflow in addition is not detected.	int64	int16x2, int16x2, int64	None
__sm1aldx	Exchange halfwords of second operand, perform 2x16-bit multiplication, adding both results to third operand. Overflow in addition is not detected.	int64	int16x2, int16x2, int64	None
__sm1sd	2x16-bit signed multiplications. Take difference of products, subtract high halfword product from low halfword product, add difference to third operand.	int32	int16x2, int16x2, int32	Q bit
__sm1sdx	Exchange halfwords of second operand, then 2x16-bit signed multiplications. Product difference is added to a third accumulate operand.	int32	int16x2, int16x2, int32	Q bit
__sm1sld	2x16-bit signed multiplications. Take difference of products, subtracting high halfword product from low halfword product, and add difference to third operand. Overflow in addition is not detected.	int64	int16x2, int16x2, int64	None
__sm1sldx	Exchange halfwords of second operand, then 2x16-bit signed multiplications. Take difference of products, subtracting high halfword product from low halfword product, and add difference to third operand. Overflow in addition is not detected.	int64	int16x2, int16x2, u64	None
__smuad	2x16-bit signed multiplications, adding the products together.	int32	int16x2, int16x2	Q bit
__smusd	2x16-bit signed multiplications. Take difference of products, subtracting high halfword product from low halfword product.	int32	int16x2, int16x2	None
__smusdx	2x16-bit signed multiplications. Product of high halfword of first operand and low halfword of second operand is subtracted from product of low halfword of first operand and high halfword of second operand, and difference is added to third operand.	int32	int16x2, int16x2	None
__ssat16	2x16-bit signed saturation to a selected width.	int16x2	int16x2, /*constant* / unsigned int	Q bit
__ssax	Exchange halfwords of second operand, subtract high halfwords and add low halfwords.	int16x2	int16x2, int16x2	APSR.GE bits

Table A-2 ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags (continued)

Intrinsic	Summary description	Byte lanes		Affected flags
		Returns	Operands	
__ssub16	2x16-bit signed subtraction.	int16x2	int16x2, int16x2	APSR.GE bits
__ssub8	4x8-bit signed subtraction.	int8x4	int8x4	APSR.GE bits
__smuadx	Exchange halfwords of second operand, perform 2x16-bit signed multiplications, and add products together.	int32	int16x2, int16x2	Q bit
__sxtab16	Two values at bit positions [23:16][7:0] are extracted from second operand, sign-extended to 16 bits, and added to first operand.	int16x2	int8x4, int16x2	None
__sxtb16	Two values at bit positions [23:16][7:0] are extracted from the operand and sign-extended to 16 bits.	int16x2	int8x4	None
__uadd16	2x16-bit unsigned addition.	uint16x2	uint16x2, uint16x2	APSR.GE bits
__uadd8	4x8-bit unsigned addition.	uint8x4	uint8x4, uint8x4	APSR.GE bits
__uasx	Exchange halfwords of second operand, add high halfwords and subtract low halfwords.	uint16x2	uint16x2, uint16x2	APSR.GE bits
__uhadd16	2x16-bit unsigned addition, halving the results.	uint16x2	uint16x2, uint16x2	None
__uhadd8	4x8-bit unsigned addition, halving the results.	uint8x4	uint8x4, uint8x4	None
__uhasx	Exchange halfwords of second operand, add high halfwords and subtract low halfwords, halving the results.	uint16x2	uint16x2, uint16x2	None
__uhsax	Exchange halfwords of second operand, subtract high halfwords and add low halfwords, halving the results.	uint16x2	uint16x2, uint16x2	None
__uhsub16	2x16-bit unsigned subtraction, halving the results.	uint16x2	uint16x2, uint16x2	None
__uhsub8	4x8-bit unsigned subtraction, halving the results.	uint8x4	uint8x4	None
__uqadd16	2x16-bit unsigned addition, saturating to range $0 \leq x \leq 2^{16} - 1$.	uint16x2	uint16x2, uint16x2	None
__uqadd8	4x8-bit unsigned addition, saturating to range $0 \leq x \leq 2^8 - 1$.	uint8x4	uint8x4, uint8x4	None
__uqasx	Exchange halfwords of second operand, perform saturating unsigned addition on high halfwords and saturating unsigned subtraction on low halfwords.	uint16x2	uint16x2, uint16x2	None
__uqsax	Exchange halfwords of second operand, perform saturating unsigned subtraction on high halfwords and saturating unsigned addition on low halfwords.	uint16x2	uint16x2, uint16x2	None
__uqsub16	2x16-bit unsigned subtraction, saturating to range $0 \leq x \leq 2^{16} - 1$.	uint16x2	uint16x2, uint16x2	None

Table A-2 ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags (continued)

Intrinsic	Summary description	Byte lanes		Affected flags
		Returns	Operands	
__uqsub8	4x8-bit unsigned subtraction, saturating to range $0 \leq x \leq 2^8 - 1$.	uint8x4	uint8x4, uint8x4	None
__usad8	4x8-bit unsigned subtraction, add absolute values of the differences together, return result as single unsigned integer.	uint32	uint8x4, uint8x4	None
__usada8	4x8-bit unsigned subtraction, add absolute values of the differences together, and add result to third operand.	uint32	uint8x4, uint8x4, uint32	None
__usax	Exchange halfwords of second operand, subtract high halfwords and add low halfwords.	uint16x2	uint16x2, uint16x2	APSR.GE bits
__usat16	Saturate two 16-bit values to a selected unsigned range. Input values are signed and output values are non-negative.	int16x2	int16x2, /*constant*/ / unsigned int	Q flag
__usub16	2x16-bit unsigned subtraction.	uint16x2	uint16x2, uint16x2	APSR.GE bits
__usub8	4x8-bit unsigned subtraction.	uint8x4	uint8x4, uint8x4	APSR.GE bits
__uxtab16	Two values at bit positions [23:16][7:0] are extracted from the second operand, zero-extended to 16 bits, and added to the first operand.	uint16x2	uint8x4, uint16x2	None
__uxtb16	Two values at bit positions [23:16][7:0] are extracted from the operand and zero-extended to 16 bits.	uint16x2	uint8x4	None

A.3 ARMv6 SIMD intrinsics, compatible processors and architectures

[Table A-3](#) lists some ARMv6 SIMD instruction intrinsics and compatible processors and architectures, as examples of compatibility.

Use of intrinsics that are not available on your target platform results in linkage failure with undefined symbols.

Table A-3 ARMv6 SIMD intrinsics, compatible processors and architectures

Intrinsics	Compatible <code>--cpu</code> options
<code>__qadd16</code> , <code>__qadd8</code> , <code>__qasx</code>	6, 6K, 6T2, 6Z, 7-A, 7-R, 7-A.security, Cortex-R4, Cortex-R4F, Cortex-R7, Cortex-R7.no_vfp, Cortex-A5, Cortex-A8, Cortex-A8.no_neon, Cortex-A8NoNEON, Cortex-A9, Cortex-A9.no_neon, Cortex-A9.no_neon.no_vfp, Cortex-A15, Cortex-A15.no_neon, Cortex-A15.no_neon.no_vfp, Cortex-A7, Cortex-A7.no_neon, Cortex-A7.no_neon.no_vfp, Cortex-M4, Cortex-M4.fp, ARM1136J-S, ARM1136JF-S, ARM1136J-S-rev1, ARM1136JF-S-rev1, ARM1156T2-S, ARM1156T2F-S, ARM1176JZ-S, ARM1176JZF-S, MPCore, MPCore.no_vfp, MPCoreNoVFP, 88FR111, 88FR111.no_hw_divide, QSP, QSP.no_neon, QSP.no_neon.no_vfp

A.3.1 See also

Reference

- [--cpu=list](#) on page 3-48
- [--cpu=name](#) on page 3-49.

A.4 ARMv6 SIMD instruction intrinsics and APSR GE flags

Table A-4 ARMv6 SIMD instruction intrinsics and APSR GE flags

Intrinsic	APSR.GE flag action	APSR.GE operation
__se1	Reads GE flags	if APSR.GE[0] == 1 then res[7:0] = val1[7:0] else val2[7:0] if APSR.GE[1] == 1 then res[15:8] = val1[15:8] else val2[15:8] if APSR.GE[2] == 1 then res[23:16] = val1[23:16] else val2[23:16] if APSR.GE[3] == 1 then res[31:24] = val1[31:24] else val2[31:24]
__sadd16	Sets or clears GE flags	if sum1 • 0 then APSR.GE[1:0] = 11 else 00 if sum2 • 0 then APSR.GE[3:2] = 11 else 00
__sadd8	Sets or clears GE flags	if sum1 • 0 then APSR.GE[0] = 1 else 0 if sum2 • 0 then APSR.GE[1] = 1 else 0 if sum3 • 0 then APSR.GE[2] = 1 else 0 if sum4 • 0 then APSR.GE[3] = 1 else 0
__sasx	Sets or clears GE flags	if diff • 0 then APSR.GE[1:0] = 11 else 00 if sum • 0 then APSR.GE[3:2] = 11 else 00
__ssax	Sets or clears GE flags	if sum • 0 then APSR.GE[1:0] = 11 else 00 if diff • 0 then APSR.GE[3:2] = 11 else 00
__ssub16	Sets or clears GE flags	if diff1 • 0 then APSR.GE[1:0] = 11 else 00 if diff2 • 0 then APSR.GE[3:2] = 11 else 00
__ssub8	Sets or clears GE flags	if diff1 • 0 then APSR.GE[0] = 1 else 0 if diff2 • 0 then APSR.GE[1] = 1 else 0 if diff3 • 0 then APSR.GE[2] = 1 else 0 if diff4 • 0 then APSR.GE[3] = 1 else 0
__uadd16	Sets or clears GE flags	if sum1 • 0x10000 then APSR.GE[1:0] = 11 else 00 if sum2 • 0x10000 then APSR.GE[3:2] = 11 else 00
__uadd8	Sets or clears GE flags	if sum1 • 0x100 then APSR.GE[0] = 1 else 0 if sum2 • 0x100 then APSR.GE[1] = 1 else 0 if sum3 • 0x100 then APSR.GE[2] = 1 else 0 if sum4 • 0x100 then APSR.GE[3] = 1 else 0
__uasx	Sets or clears GE flags	if diff • 0 then APSR.GE[1:0] = 11 else 00 if sum • 0x10000 then APSR.GE[3:2] = 11 else 00
__usax	Sets or clears GE flags	if sum • 0x10000 then APSR.GE[1:0] = 11 else 00 if diff • 0 then APSR.GE[3:2] = 11 else 00
__usub16	Sets or clears GE flags	if diff1 • 0 then APSR.GE[1:0] = 11 else 00 if diff2 • 0 then APSR.GE[3:2] = 11 else 00
__usub8	Sets or clears GE flags	if diff1 • 0 then APSR.GE[0] = 1 else 0 if diff2 • 0 then APSR.GE[1] = 1 else 0 if diff3 • 0 then APSR.GE[2] = 1 else 0 if diff4 • 0 then APSR.GE[3] = 1 else 0

A.5 `__qadd16` intrinsic

This intrinsic inserts a QADD16 instruction into the instruction stream generated by the compiler. It enables you to perform two 16-bit integer arithmetic additions in parallel, saturating the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$.

```
unsigned int __qadd16(unsigned int va1, unsigned int va2)
```

Where:

`va1` holds the first two 16-bit summands

`va2` holds the second two 16-bit summands.

The `__qadd16` intrinsic returns:

- the saturated addition of the low halfwords in the low halfword of the return value
- the saturated addition of the high halfwords in the high halfword of the return value.

The returned results are saturated to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$.

Example:

```
unsigned int add_halfwords(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __qadd16(va1, va2); /* res[15:0] = va1[15:0] + va2[15:0]
                               res[16:31] = va1[31:16] + va2[31:16]
                               */
    return res;
}
```

A.5.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Saturating instructions on page 3-19 in the Assembler Reference](#)
- [QADD, QSUB, QDADD, and QDSUB on page 3-119 in the Assembler Reference.](#)

A.6 __qadd8 intrinsic

This intrinsic inserts a QADD8 instruction into the instruction stream generated by the compiler. It enables you to perform four 8-bit integer additions, saturating the results to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$.

```
unsigned int __qadd8(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first four 8-bit summands
va2 holds the other four 8-bit summands.

The __qadd8 intrinsic returns:

- the saturated addition of the first byte of each operand in the first byte of the return value
- the saturated addition of the second byte of each operand in the second byte of the return value
- the saturated addition of the third byte of each operand in the third byte of the return value
- the saturated addition of the fourth byte of each operand in the fourth byte of the return value.

The returned results are saturated to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$.

Example:

```
unsigned int add_bytes(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __qadd8(va1, va2); /* res[7:0] = va1[7:0] + va2[7:0]
                             res[15:8] = va1[15:8] + va2[15:8]
                             res[23:16] = va1[23:16] + va2[23:16]
                             res[31:24] = va1[31:24] + va2[31:24]
                             */
    return res;
}
```

A.6.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Saturating instructions](#) on page 3-19 in the *Assembler Reference*
- [QADD, QSUB, QDADD, and QDSUB](#) on page 3-119 in the *Assembler Reference*.

A.7 __qasx intrinsic

This intrinsic inserts a QASX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of the one operand, then add the high halfwords and subtract the low halfwords, saturating the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$.

```
unsigned int __qasx(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first operand for the subtraction in the low halfword, and the first operand for the addition in the high halfword

va2 holds the second operand for the subtraction in the high halfword, and the second operand for the addition in the low halfword.

The __qasx intrinsic returns:

- the saturated subtraction of the high halfword in the second operand from the low halfword in the first operand, in the low halfword of the return value
- the saturated addition of the high halfword in the first operand and the low halfword in the second operand, in the high halfword of the return value.

The returned results are saturated to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$.

Example:

```
unsigned int exchange_add_and_subtract(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __qasx(va1,va2); /* res[15:0] = va1[15:0] - va2[31:16]
                          res[31:16] = va1[31:16] + va2[15:0]
                          */
                          /* Alternative equivalent representation:
                          va2[15:0][31:16] = va2[31:16][15:0]
                          res[15:0] = va1[15:0] - va2[15:0]
                          res[31:16] = va1[31:16] + va2[31:16]
                          */

    return res;
}
```

A.7.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Saturating instructions](#) on page 3-19 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.8 __qsax intrinsic

This intrinsic inserts a QSAX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of one operand, then subtract the high halfwords and add the low halfwords, saturating the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$.

```
unsigned int __qsax(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first operand for the addition in the low halfword, and the first operand for the subtraction in the high halfword

va2 holds the second operand for the addition in the high halfword, and the second operand for the subtraction in the low halfword.

The __qsax intrinsic returns:

- the saturated addition of the low halfword of the first operand and the high halfword of the second operand, in the low halfword of the return value
- the saturated subtraction of the low halfword of the second operand from the high halfword of the first operand, in the high halfword of the return value.

The returned results are saturated to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$.

Example:

```
unsigned int exchange_subtract_and_add(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __qsax(va1,va2); /* res[15:0] = va1[15:0] + va2[31:16]
                           res[31:16] = va1[31:16] - va2[15:0]
                           */
                           /* Alternative equivalent representation:
                           va2[15:0][31:16] = va2[31:16][15:0]
                           res[15:0] = va1[15:0] + va2[15:0]
                           res[31:16] = va1[31:16] - va2[31:16]
                           */

    return res;
}
```

A.8.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Saturating instructions](#) on page 3-19 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.9 __qsub16 intrinsic

This intrinsic inserts a QSUB16 instruction into the instruction stream generated by the compiler. It enables you to perform two 16-bit integer subtractions, saturating the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$.

```
unsigned int __qsub16(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first halfword operands

va2 holds the second halfword operands.

The __qsub16 intrinsic returns:

- the saturated subtraction of the low halfword in the second operand from the low halfword in the first operand, in the low halfword of the returned result
- the saturated subtraction of the high halfword in the second operand from the high halfword in the first operand, in the high halfword of the returned result.

The returned results are saturated to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$.

Example:

```
unsigned int subtract_halfwords(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __qsub16(va1,va2); /* res[15:0] = va1[15:0] - va2[15:0]
                           res[31:16] = va1[31:16] - va2[31:16]
                           */

    return res;
}
```

A.9.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Saturating instructions on page 3-19 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.10 __qsub8 intrinsic

This intrinsic inserts a QSUB8 instruction into the instruction stream generated by the compiler. It enables you to perform four 8-bit integer subtractions, saturating the results to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$.

```
unsigned int __qsub8(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first four 8-bit operands

va2 holds the second four 8-bit operands.

The __qsub8 intrinsic returns:

- the subtraction of the first byte in the second operand from the first byte in the first operand, in the first byte of the return value
- the subtraction of the second byte in the second operand from the second byte in the first operand, in the second byte of the return value
- the subtraction of the third byte in the second operand from the third byte in the first operand, in the third byte of the return value
- the subtraction of the fourth byte in the second operand from the fourth byte in the first operand, in the fourth byte of the return value.

The returned results are saturated to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$.

Example:

```
unsigned int subtract_bytes(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __qsub8(va1,va2); /* res[7:0] = va1[7:0] - va2[7:0]
                           res[15:8] = va1[15:8] - va2[15:8]
                           res[23:16] = va1[23:16] - va2[23:16]
                           res[31:24] = va1[31:24] - va2[31:24]
                           */
    return res;
}
```

A.10.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Saturating instructions on page 3-19 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.11 __sadd16 intrinsic

This intrinsic inserts an SADD16 instruction into the instruction stream generated by the compiler. It enables you to perform two 16-bit signed integer additions. The GE bits in the APSR are set according to the results of the additions.

```
unsigned int __sadd16(unsigned int va11, unsigned int va12)
```

Where:

va11 holds the first two 16-bit summands

va12 holds the second two 16-bit summands.

The __sadd16 intrinsic returns:

- the addition of the low halfwords in the low halfword of the return value
- the addition of the high halfwords in the high halfword of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If *res* is the return value, then:

- if $res[15:0] \geq 0$ then APSR.GE[1:0] = 11 else 00
- if $res[31:16] \geq 0$ then APSR.GE[3:2] = 11 else 00.

Example:

```
unsigned int add_halfwords(unsigned int va11, unsigned int va12)
{
    unsigned int res;

    res = __sadd16(va11,va12); /* res[15:0] = va11[15:0] + va12[15:0]
                               res[31:16] = va11[31:16] + va12[31:16]
                               */
    return res;
}
```

A.11.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [__sel intrinsic](#) on page A-20
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Saturating instructions](#) on page 3-19 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.12 __sadd8 intrinsic

This intrinsic inserts an SADD8 instruction into the instruction stream generated by the compiler. It enables you to perform four 8-bit signed integer additions. The GE bits in the APSR are set according to the results of the additions.

```
unsigned int __sadd8(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first four 8-bit summands
va2 holds the second four 8-bit summands.

The __sadd8 intrinsic returns:

- the addition of the first bytes from each operand, in the first byte of the return value
- the addition of the second bytes of each operand, in the second byte of the return value
- the addition of the third bytes of each operand, in the third byte of the return value
- the addition of the fourth bytes of each operand, in the fourth byte of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If *res* is the return value, then:

- if $res[7:0] \geq 0$ then APSR.GE[0] = 1 else 0
- if $res[15:8] \geq 0$ then APSR.GE[1] = 1 else 0.
- if $res[23:16] \geq 0$ then APSR.GE[2] = 1 else 0.
- if $res[31:24] \geq 0$ then APSR.GE[3] = 1 else 0.

Example:

```
unsigned int add_bytes(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __sadd16(va1,va2); /* res[7:0] = va1[7:0] + va2[7:0]
                             res[15:8] = va1[15:8] + va2[15:8]
                             res[23:16] = va1[23:16] + va2[23:16]
                             res[31:24] = va1[31:24] + va2[31:24]
                             */
    return res;
}
```

A.12.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [__sel intrinsic](#) on page A-20
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Saturating instructions](#) on page 3-19 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.13 __sasx intrinsic

This intrinsic inserts an SASX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of the second operand, add the high halfwords and subtract the low halfwords. The GE bits in the APSR are set according to the results.

```
unsigned int __sasx(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first operand for the subtraction in the low halfword, and the first operand for the addition in the high halfword

va2 holds the second operand for the subtraction in the high halfword, and the second operand for the addition in the low halfword.

The __sasx intrinsic returns:

- the subtraction of the high halfword in the second operand from the low halfword in the first operand, in the low halfword of the return value
- the addition of the high halfword in the first operand and the low halfword in the second operand, in the high halfword of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If *res* is the return value, then:

- if $res[15:0] \geq 0$ then APSR.GE[1:0] = 11 else 00
- if $res[31:16] \geq 0$ then APSR.GE[3:2] = 11 else 00.

Example:

```
unsigned int exchange_subtract_add(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __sasx(va1, va2); /* res[15:0] = va1[15:0] - va2[31:16]
                           res[31:16] = va1[31:16] + va2[15:0]
                           */
    return res;
}
```

A.13.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [__sel intrinsic](#) on page A-20
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.14 `__sel` intrinsic

This intrinsic inserts a SEL instruction into the instruction stream generated by the compiler. It enables you to select bytes from the input parameters, whereby the bytes that are selected depend upon the results of previous SIMD instruction intrinsics. The results of previous SIMD instruction intrinsics are represented by the *Greater than or Equal* flags in the *Application Program Status Register* (APSR).

The `__sel` intrinsic works equally well on both halfword and byte operand intrinsic results. This is because halfword operand operations set two (duplicate) GE bits per value. For example, the `__sax` intrinsic.

```
unsigned int __sel(unsigned int va1, unsigned int va2)
```

Where:

`va1` holds four selectable bytes
`va2` holds four selectable bytes.

The `__sel` intrinsic selects bytes from the input parameters and returns them in the return value, `res`, according to the following criteria:

```
if APSR.GE[0] == 1 then res[7:0] = va1[7:0] else res[7:0] = va2[7:0]
if APSR.GE[1] == 1 then res[15:8] = va1[15:8] else res[15:8] = va2[15:8]
if APSR.GE[2] == 1 then res[23:16] = va1[23:16] else res[23:16] = va2[23:16]
if APSR.GE[3] == 1 then res[31:24] = va1[31:24] else res = va2[31:24]
```

Example:

```
unsigned int ge_filter(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __sel(va1,va2);
    return res;
}

unsigned int foo(unsigned int a, unsigned int b)
{
    int res;
    int filtered_res;

    res = __sax(a,b); /* This intrinsic sets the GE flags */
    filtered_res = ge_filter(res); /* Filter the results of the __sax */
                                /* intrinsic. Some results are filtered */
                                /* out based on the GE flags. */

    return filtered_res;
}
```

A.14.1 See also

- [__sadd16 intrinsic](#) on page A-17
- [__sax intrinsic](#) on page A-19
- [__ssax intrinsic](#) on page A-40
- [__ssub8 intrinsic](#) on page A-42
- [__ssub16 intrinsic](#) on page A-41
- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SEL](#) on page 3-127 in the *Assembler Reference*.

A.15 __shadd16 intrinsic

This intrinsic inserts a SHADD16 instruction into the instruction stream generated by the compiler. It enables you to perform two signed 16-bit integer additions, halving the results.

```
unsigned int __shadd16(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first two 16-bit summands

va2 holds the second two 16-bit summands.

The __shadd16 intrinsic returns:

- the halved addition of the low halfwords from each operand, in the low halfword of the return value
- the halved addition of the high halfwords from each operand, in the high halfword of the return value.

Example:

```
unsigned int add_and_halfve(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __shadd16(va1,va2); /* res[15:0] = (va1[15:0] + va2[15:0]) >> 1
                               res[31:16] = (va1[31:16] + va2[31:16]) >> 1
                               */
    return res;
}
```

A.15.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.16 __shadd8 intrinsic

This intrinsic inserts a SHADD8 instruction into the instruction stream generated by the compiler. It enables you to perform four signed 8-bit integer additions, halving the results.

```
unsigned int __shadd8(unsigned int va11, unsigned int va12)
```

Where:

va11 holds the first four 8-bit summands
va12 holds the second four 8-bit summands.

The __shadd8 intrinsic returns:

- the halved addition of the first bytes from each operand, in the first byte of the return value
- the halved addition of the second bytes from each operand, in the second byte of the return value
- the halved addition of the third bytes from each operand, in the third byte of the return value
- the halved addition of the fourth bytes from each operand, in the fourth byte of the return value.

Example:

```
unsigned int add_and_halve(unsigned int va11, unsigned int va12)
{
    unsigned int res;

    res = __shadd8(va11,va12); /* res[7:0] = (va11[7:0] + va12[7:0]) >> 1
                               res[15:8] = (va11[15:8] + va12[15:8]) >> 1
                               res[23:16] = (va11[23:16] + va12[23:16]) >> 1
                               res[31:24] = (va11[31:24] + va12[31:24]) >> 1
                               */
    return res;
}
```

A.16.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.17 __shasx intrinsic

This intrinsic inserts a SHASX instruction into the instruction stream generated by the compiler. It enables you to exchange the two halfwords of one operand, perform one signed 16-bit integer addition and one signed 16-bit subtraction, and halve the results.

```
unsigned int __shasx(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first halfword operands

va2 holds the second halfword operands.

The __shasx intrinsic returns:

- the halved subtraction of the high halfword in the second operand from the low halfword in the first operand, in the low halfword of the return value
- the halved subtraction of the low halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

Example:

```
unsigned int exchange_add_subtract_half(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __shasx(va1,va2); /* res[15:0] = (va1[15:0] - va2[31:16]) >> 1
                           res[31:16] = (va1[31:16] - va2[15:0]) >> 1
                           */
    return res;
}
```

A.17.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.18 __shsax intrinsic

This intrinsic inserts a SHSAX instruction into the instruction stream generated by the compiler. It enables you to exchange the two halfwords of one operand, perform one signed 16-bit integer subtraction and one signed 16-bit addition, and halve the results.

```
unsigned int __shsax(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first halfword operands

va2 holds the second halfword operands.

The __shsax intrinsic returns:

- the halved addition of the low halfword in the first operand and the high halfword in the second operand, in the low halfword of the return value
- the halved subtraction of the low halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

Example:

```
unsigned int exchange_subtract_add_halve(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __shsax(va1,va2); /* res[15:0] = (va1[15:0] + va2[31:16]) >> 1
                           res[31:16] = (va1[31:16] - va2[15:0]) >> 1
                           */
    return res;
}
```

A.18.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.19 __shsub16 intrinsic

This intrinsic inserts a SHSUB16 instruction into the instruction stream generated by the compiler. It enables you to perform two signed 16-bit integer subtractions, halving the results.

```
unsigned int __shsub16(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first halfword operands

va2 holds the second halfword operands.

The __shsub16 intrinsic returns:

- the halved subtraction of the low halfword in the second operand from the low halfword in the first operand, in the low halfword of the return value
- the halved subtraction of the high halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

Example:

```
unsigned int add_and_halfve(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __shsub16(va1,va2); /* res[15:0] = (va1[15:0] - va2[15:0]) >> 1
                             res[31:16] = (va1[31:16] - va2[31:16]) >> 1
                             */
    return res;
}
```

A.19.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.20 __shsub8 intrinsic

This intrinsic inserts a SHSUB8 instruction into the instruction stream generated by the compiler. It enables you to perform four signed 8-bit integer subtractions, halving the results.

```
unsigned int __shsub8(unsigned int va11, unsigned int va12)
```

Where:

va11 holds the first four operands
va12 holds the second four operands.

The __shsub8 intrinsic returns:

- the halved subtraction of the first byte in the second operand from the first byte in the first operand, in the first byte of the return value
- the halved subtraction of the second byte in the second operand from the second byte in the first operand, in the second byte of the return value
- the halved subtraction of the third byte in the second operand from the third byte in the first operand, in the third byte of the return value
- the halved subtraction of the fourth byte in the second operand from the fourth byte in the first operand, in the fourth byte of the return value

Example:

```
unsigned int subtract_and_halve(unsigned int va11, unsigned int va12)
{
    unsigned int res;

    res = __shsub8(va11,va12); /* res[7:0] = (va11[7:0] - va12[7:0]) >> 1
                               res[15:8] = (va11[15:8] - va12[15:8]) >> 1
                               res[23:16] = (va11[23:16] - va12[23:16]) >> 1
                               res[31:24] = (va11[31:24] - va12[31:24]) >> 1
                               */
    return res;
}
```

A.20.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.21 __smlad intrinsic

This intrinsic inserts an SMLAD instruction into the instruction stream generated by the compiler. It enables you to perform two signed 16-bit multiplications, adding both results to a 32-bit accumulate operand. The Q bit is set if the addition overflows. Overflow cannot occur during the multiplications.

```
unsigned int __smlad(unsigned int va1, unsigned int va2, unsigned int va3)
```

Where:

va1 holds the first halfword operands for each multiplication
va2 holds the second halfword operands for each multiplication
va3 holds the accumulate value.

The __smlad intrinsic returns the product of each multiplication added to the accumulate value, as a 32-bit integer.

Example:

```
unsigned int dual_multiply_accumulate(unsigned int va1, unsigned int va2, unsigned
int va3)
{
    unsigned int res;

    res = __smlad(va1,va2,va3); /* p1 = va1[15:0] x va2[15:0]
                               p2 = va1[31:16] x va2[31:16]
                               res[31:0] = p1 + p2 + va3[31:0]
                               */
    return res;
}
```

A.21.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SMLAD and SMLSD](#) on page 3-133 in the *Assembler Reference*.

A.22 __smladx intrinsic

This intrinsic inserts an SMLADX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of the second operand, perform two signed 16-bit multiplications, adding both results to a 32-bit accumulate operand. The Q bit is set if the addition overflows. Overflow cannot occur during the multiplications.

```
unsigned int __smladx(unsigned int va11, unsigned int va12, unsigned int va13)
```

Where:

va11 holds the first halfword operands for each multiplication
va12 holds the second halfword operands for each multiplication
va13 holds the accumulate value.

The __smladx intrinsic returns the product of each multiplication added to the accumulate value, as a 32-bit integer.

Example:

```
unsigned int dual_multiply_accumulate(unsigned int va11, unsigned int va12, unsigned
int va13)
{
    unsigned int res;

    res = __smladx(va11, va12, va13); /* p1 = va11[15:0] x va12[31:16]
                                     p2 = va11[31:16] x va12[15:0]
                                     res[31:0] = p1 + p2 + va13[31:0]
                                     */
    return res;
}
```

A.22.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [SMLAD and SMLSD on page 3-133 in the Assembler Reference.](#)

A.23 __smlald intrinsic

This intrinsic inserts an SMLALD instruction into the instruction stream generated by the compiler. It enables you to perform two signed 16-bit multiplications, adding both results to a 64-bit accumulate operand. Overflow is only possible as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64} .

```
unsigned long long __smlald(unsigned int va1, unsigned int va2, unsigned long long va3)
```

Where:

<i>va1</i>	holds the first halfword operands for each multiplication
<i>va2</i>	holds the second halfword operands for each multiplication
<i>va3</i>	holds the accumulate value.

The __smlald intrinsic returns the product of each multiplication added to the accumulate value.

Example:

```
unsigned int dual_multiply_accumulate(unsigned int va1, unsigned int va2, unsigned
int va3)
{
    unsigned int res;

    res = __smlald(va1,va2,va3); /* p1 = va1[15:0] x va2[15:0]
                                p2 = va1[31:16] x va2[31:16]
                                sum = p1 + p2 + va3[63:32][31:0]
                                res[63:32] = sum[63:32]
                                res[31:0] = sum[31:0]
                                */

    return res;
}
```

A.23.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [SMLALD and SMLSLD on page 3-137 in the Assembler Reference.](#)

A.24 __smlaldx intrinsic

This intrinsic inserts an SMLALDX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of the second operand, and perform two signed 16-bit multiplications, adding both results to a 64-bit accumulate operand. Overflow is only possible as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64} .

```
unsigned long long __smlaldx(unsigned int va1, unsigned int va2, unsigned long long va3)
```

Where:

va1 holds the first halfword operands for each multiplication
va2 holds the second halfword operands for each multiplication
va3 holds the accumulate value.

The __smlald intrinsic returns the product of each multiplication added to the accumulate value.

Example:

```
unsigned int dual_multiply_accumulate(unsigned int va1, unsigned int va2, unsigned
int va3)
{
    unsigned int res;

    res = __smlald(va1,va2,va3); /* p1 = va1[15:0] x va2[31:16]
                                p2 = va1[31:16] x va2[15:0]
                                sum = p1 + p2 + va3[63:32][31:0]
                                res[63:32] = sum[63:32]
                                res[31:0] = sum[31:0]
                                */

    return res;
}
```

A.24.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SMLALD and SMLSLD](#) on page 3-137 in the *Assembler Reference*.

A.25 __sm1sd intrinsic

This intrinsic inserts an SMLSD instruction into the instruction stream generated by the compiler. It enables you to perform two 16-bit signed multiplications, take the difference of the products, subtracting the high halfword product from the low halfword product, and add the difference to a 32-bit accumulate operand. The Q bit is set if the accumulation overflows. Overflow cannot occur during the multiplications or the subtraction.

```
unsigned int __sm1sd(unsigned int va1, unsigned int va2, unsigned int va3)
```

Where:

va1 holds the first halfword operands for each multiplication
va2 holds the second halfword operands for each multiplication
va3 holds the accumulate value.

The __sm1sd intrinsic returns the difference of the product of each multiplication, added to the accumulate value.

Example:

```
unsigned int dual_multiply_diff_prods(unsigned int va1, unsigned int va2, unsigned
int va3)
{
    unsigned int res;

    res = __sm1sd(va1,va2,va3); /* p1 = va1[15:0] x va2[15:0]
                               p2 = va1[31:16] x va2[31:16]
                               res[31:0] = p1 - p2 + va3[31:0]
                               */
    return res;
}
```

A.25.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SMLAD and SMLSD](#) on page 3-133 in the *Assembler Reference*.

A.26 __smlsdx intrinsic

This intrinsic inserts an SMLSDX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords in the second operand, then perform two 16-bit signed multiplications. The difference of the products is added to a 32-bit accumulate operand. The Q bit is set if the addition overflows. Overflow cannot occur during the multiplications or the subtraction.

```
unsigned int __smlsdx(unsigned int va1, unsigned int va2, unsigned int va3)
```

Where:

va1 holds the first halfword operands for each multiplication
va2 holds the second halfword operands for each multiplication
va3 holds the accumulate value.

The __smlsd intrinsic returns the difference of the product of each multiplication, added to the accumulate value.

Example:

```
unsigned int dual_multiply_diff_prods(unsigned int va1, unsigned int va2, unsigned
int va3)
{
    unsigned int res;

    res = __smlsd(va1,va2,va3); /* p1 = va1[15:0] x va2[31:16]
                               p2 = va1[31:16] x va2[15:0]
                               res[31:0] = p1 - p2 + va3[31:0]
                               */
    return res;
}
```

A.26.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SMLAD and SMLSD](#) on page 3-133 in the *Assembler Reference*.

A.27 __sm1s1d intrinsic

This intrinsic inserts an SMLS�D instruction into the instruction stream generated by the compiler. It enables you to perform two 16-bit signed multiplications, take the difference of the products, subtracting the high halfword product from the low halfword product, and add the difference to a 64-bit accumulate operand. Overflow cannot occur during the multiplications or the subtraction. Overflow can occur as a result of the 64-bit addition, and this overflow is not detected. Instead, the result wraps round to modulo 2^{64} .

```
unsigned long long __sm1s1d(unsigned int va11, unsigned int va12, unsigned long long va13)
```

Where:

va11 holds the first halfword operands for each multiplication
va12 holds the second halfword operands for each multiplication
va13 holds the accumulate value.

The `__sm1s1d` intrinsic returns the difference of the product of each multiplication, added to the accumulate value.

Example:

```
unsigned long long dual_multiply_diff_prods(unsigned int va11, unsigned int va12,
unsigned long long va13)
{
    unsigned int res;

    res = __sm1s1d(va11,va12,va13); /* p1 = va11[15:0] x va12[15:0]
                                   p2 = va11[31:16] x va12[31:16]
                                   res[63:0] = p1 - p2 + va13[63:0]
                                   */
    return res;
}
```

A.27.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SMLALD and SMLS�D](#) on page 3-137 in the *Assembler Reference*.

A.28 __sm1s1dx intrinsic

This intrinsic inserts an SMLSXD instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of the second operand, perform two 16-bit multiplications, adding the difference of the products to a 64-bit accumulate operand. Overflow cannot occur during the multiplications or the subtraction. Overflow can occur as a result of the 64-bit addition, and this overflow is not detected. Instead, the result wraps round to modulo 2^{64} .

```
unsigned long long __sm1s1dx(unsigned int va1, unsigned int va2, unsigned long long va3)
```

Where:

va1 holds the first halfword operands for each multiplication
va2 holds the second halfword operands for each multiplication
va3 holds the accumulate value.

The __sm1s1d intrinsic returns the difference of the product of each multiplication, added to the accumulate value.

Example:

```
unsigned long long dual_multiply_diff_prods(unsigned int va1, unsigned int va2,
unsigned long long va3)
{
    unsigned int res;

    res = __sm1s1d(va1,va2,va3); /* p1 = va1[15:0] x va2[31:16]
                                p2 = va1[31:16] x va2[15:0]
                                res[63:0] = p1 - p2 + va3[63:0]
                                */

    return res;
}
```

A.28.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [SMLALD and SMLSLD on page 3-137 in the Assembler Reference.](#)

A.29 __smuad intrinsic

This intrinsic inserts an SMUAD instruction into the instruction stream generated by the compiler. It enables you to perform two 16-bit signed multiplications, adding the products together. The Q bit is set if the addition overflows.

```
unsigned int __smuad(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first halfword operands for each multiplication

va2 holds the second halfword operands for each multiplication.

The __smuad intrinsic returns the products of the two 16-bit signed multiplications.

Example:

```
unsigned int dual_multiply_prods(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __smuad(va1, va2); /* p1 = va1[15:0] x va2[15:0]
                             p2 = va1[31:16] x va2[31:16]
                             res[31:0] = p1 + p2
                             */
    return res;
}
```

A.29.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SMUAD{X} and SMUSD{X}](#) on page 3-141 in the *Assembler Reference*.

A.30 __smuadx intrinsic

This intrinsic inserts an SMUADX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of the second operand, perform two 16-bit signed integer multiplications, and add the products together. Exchanging the halfwords of the second operand produces top \times bottom and bottom \times top multiplication. The Q flag is set if the addition overflows. The multiplications cannot overflow.

```
unsigned int __smuadx(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first halfword operands for each multiplication
va2 holds the second halfword operands for each multiplication.

The __smuadx intrinsic returns the products of the two 16-bit signed multiplications.

Example:

```
unsigned int exchange_dual_multiply_prods(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __smuadx(va1, va2); /* va2[31:16][15:0] = va2[15:0][31:16]
                             p1 = va1[15:0] x va2[15:0]
                             p2 = va1[31:16] x va2[31:16]
                             res[31:0] = p1 + p2
                             */
    return res;
}
```

A.30.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SMUAD{X} and SMUSD{X}](#) on page 3-141 in the *Assembler Reference*.

A.31 __smusd intrinsic

This intrinsic inserts an SMUSD instruction into the instruction stream generated by the compiler. It enables you to perform two 16-bit signed multiplications, taking the difference of the products by subtracting the high halfword product from the low halfword product.

```
unsigned int __smusd(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first halfword operands for each multiplication

va2 holds the second halfword operands for each multiplication.

The __smusd intrinsic returns the difference of the products of the two 16-bit signed multiplications.

Example:

```
unsigned int dual_multiply_prods(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __smuad(va1, va2); /* p1 = va1[15:0] x va2[15:0]
                           p2 = va1[31:16] x va2[31:16]
                           res[31:0] = p1 - p2
                           */
    return res;
}
```

A.31.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SMUAD{X} and SMUSD{X}](#) on page 3-141 in the *Assembler Reference*.

A.32 __smusdx intrinsic

This intrinsic inserts an SMUSD X instruction into the instruction stream generated by the compiler. It enables you to perform two 16-bit signed multiplications, subtracting one of the products from the other. The halfwords of the second operand are exchanged before performing the arithmetic. This produces top \times bottom and bottom \times top multiplication.

```
unsigned int __smusdx(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first halfword operands for each multiplication

va2 holds the second halfword operands for each multiplication.

The __smusdx intrinsic returns the difference of the products of the two 16-bit signed multiplications.

Example:

```
unsigned int dual_multiply_prods(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __smuad(va1, va2); /* p1 = va1[15:0] x va2[31:16]
                           p2 = va1[31:16] x va2[15:0]
                           res[31:0] = p1 - p2
                           */
    return res;
}
```

A.32.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SMUAD{X} and SMUSD{X}](#) on page 3-141 in the *Assembler Reference*.

A.33 __ssat16 intrinsic

This intrinsic inserts an SSAT16 instruction into the instruction stream generated by the compiler. It enables you to saturate two signed 16-bit values to a selected signed range.

The Q bit is set if either operation saturates.

```
unsigned int __saturate_halfwords(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the two signed 16-bit values to be saturated

va2 is the bit position for saturation, an integral constant expression in the range 1 to 16.

The __ssat16 intrinsic returns:

- the signed saturation of the low halfword in *va1*, saturated to the bit position specified in *va2* and returned in the low halfword of the return value
- the signed saturation of the high halfword in *va1*, saturated to the bit position specified in *va2* and returned in the high halfword of the return value.

Example:

```
unsigned int saturate_halfwords(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __ssat16(va1,va2); /* Saturate halfwords in va1 to the signed
                               range specified by the bit position in va2 */
    return res;
}
```

A.33.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Saturating instructions on page 3-19 in the Assembler Reference](#)
- [SSAT16 and USAT16 on page 3-150 in the Assembler Reference.](#)

A.34 __ssax intrinsic

This intrinsic inserts an SSAX instruction into the instruction stream generated by the compiler. It enables you to exchange the two halfwords of one operand and perform one 16-bit integer subtraction and one 16-bit addition.

The GE bits in the APSR are set according to the results.

```
unsigned int __ssax(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first operand for the addition in the low halfword, and the first operand for the subtraction in the high halfword

va2 holds the second operand for the addition in the high halfword, and the second operand for the subtraction in the low halfword.

The __ssax intrinsic returns:

- the addition of the low halfword in the first operand and the high halfword in the second operand, in the low halfword of the return value
- the subtraction of the low halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If *res* is the return value, then:

- if $res[15:0] \geq 0$ then $APSR.GE[1:0] = 11$ else 00
- if $res[31:16] \geq 0$ then $APSR.GE[3:2] = 11$ else 00.

Example:

```
unsigned int exchange_subtract_add(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __ssax(va1,va2); /* res[15:0] = va1[15:0] + va2[31:16]
                          res[31:16] = va1[31:16] - va2[15:0]
                          */
    return res;
}
```

A.34.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.35 __ssub16 intrinsic

This intrinsic inserts an SSUB16 instruction into the instruction stream generated by the compiler. It enables you to perform two 16-bit signed integer subtractions.

The GE bits in the APSR are set according to the results.

```
unsigned int __ssub16(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first operands of each subtraction in the low and the high halfwords

va2 holds the second operands for each subtraction in the low and the high halfwords.

The __ssub16 intrinsic returns:

- the subtraction of the low halfword in the second operand from the low halfword in the first operand, in the low halfword of the return value
- the subtraction of the high halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If *res* is the return value, then:

- if $res[15:0] \geq 0$ then $APSR.GE[1:0] = 11$ else 00
- if $res[31:16] \geq 0$ then $APSR.GE[3:2] = 11$ else 00.

Example:

```
unsigned int subtract_halfwords(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __ssub16(va1, va2); /* res[15:0] = va1[15:0] - va2[15:0]
                             res[31:16] = va1[31:16] - va2[31:16]
                             */
    return res;
}
```

A.35.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [__sel intrinsic](#) on page A-20
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.36 __ssub8 intrinsic

This intrinsic inserts an SSUB8 instruction into the instruction stream generated by the compiler. It enables you to perform four 8-bit signed integer subtractions.

The GE bits in the APSR are set according to the results.

```
unsigned int __ssub8(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first four 8-bit operands of each subtraction
va2 holds the second four 8-bit operands of each subtraction.

The __ssub8 intrinsic returns:

- the subtraction of the first byte in the second operand from the first byte in the first operand, in the first bytes of the return value
- the subtraction of the second byte in the second operand from the second byte in the first operand, in the second byte of the return value
- the subtraction of the third byte in the second operand from the third byte in the first operand, in the third byte of the return value
- the subtraction of the fourth byte in the second operand from the fourth byte in the first operand, in the fourth byte of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If *res* is the return value, then:

- if $res[8:0] \geq 0$ then APSR.GE[0] = 1 else 0
- if $res[15:8] \geq 0$ then APSR.GE[1] = 1 else 0
- if $res[23:16] \geq 0$ then APSR.GE[2] = 1 else 0
- if $res[31:24] \geq 0$ then APSR.GE[3] = 1 else 0.

Example:

```
unsigned int subtract_bytes(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __ssub8(va1,va2); /* res[7:0] = va1[7:0] - va2[7:0]
                           res[15:8] = va1[15:8] - va2[15:8]
                           res[23:16] = va1[23:16] - va2[23:16]
                           res[31:24] = va1[31:24] - va2[31:24]
                           */
    return res;
}
```

A.36.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [__sel intrinsic](#) on page A-20
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.37 __sxtab16 intrinsic

This intrinsic inserts an SXTAB16 instruction into the instruction stream generated by the compiler. It enables you to extract two 8-bit values from the second operand (at bit positions [7:0] and [23:16]), sign-extend them to 16-bits each, and add the results to the first operand.

```
unsigned int __sxtab16(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the values that the extracted and sign-extended values are added to
va2 holds the two 8-bit values to be extracted and sign-extended.

The __sxtab16 intrinsic returns the addition of *va1* and *va2*, where the 8-bit values in *va2*[7:0] and *va2*[23:16] have been extracted and sign-extended prior to the addition.

Example:

```
unsigned int extract_sign_extend_and_add(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __sxtab16(va1,va2); /* res[15:0]
                             = va1[15:0] + SignExtended(va2[7:0])

                             res[31:16]
                             = va1[31:16] + SignExtended(va2[23:16])
                             */

    return res;
}
```

A.37.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SXT, SXTA, UXT, and UXTA](#) on page 3-156 in the *Assembler Reference*.

A.38 __sxtb16 intrinsic

This intrinsic inserts an SXTB16 instruction into the instruction stream generated by the compiler. It enables you to extract two 8-bit values from an operand and sign-extend them to 16 bits each.

```
unsigned int __sxtb16(unsigned int va1)
```

Where `va1[7:0]` and `va1[23:16]` hold the two 8-bit values to be sign-extended.

The `__sxtb16` intrinsic returns the 8-bit values sign-extended to 16-bit values.

Example:

```
unsigned int sign_extend(unsigned int va1)
{
    unsigned int res;

    res = __sxtb16(va1,va1); /* res[15:0] = SignExtended(va1[7:0])
                             res[31:16] = SignExtended(va1[23:16])
                             */
    return res;
}
```

A.38.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [SXT, SXTA, UXT, and UXTA on page 3-156 in the Assembler Reference.](#)

A.39 __uadd16 intrinsic

This intrinsic inserts a UADD16 instruction into the instruction stream generated by the compiler. It enables you to perform two 16-bit unsigned integer additions.

The GE bits in the APSR are set according to the results.

```
unsigned int __uadd16(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first two halfword summands for each addition
va2 holds the second two halfword summands for each addition.

The __uadd16 intrinsic returns:

- the addition of the low halfwords in each operand, in the low halfword of the return value
- the addition of the high halfwords in each operand, in the high halfword of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If *res* is the return value, then:

- if $res[15:0] \geq 0x10000$ then $APSR.GE[0] = 11$ else 00
- if $res[31:16] \geq 0x10000$ then $APSR.GE[1] = 11$ else 00.

Example:

```
unsigned int add_halfwords(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uadd16(va1,va2); /* res[15:0] = va1[15:0] + va2[15:0]
                           res[31:16] = va1[31:16] + va2[31:16]
                           */
    return res;
}
```

A.39.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.40 `__uadd8` intrinsic

This intrinsic inserts a UADD8 instruction into the instruction stream generated by the compiler. It enables you to perform four unsigned 8-bit integer additions.

The GE bits in the APSR are set according to the results.

```
unsigned int __uadd8(unsigned int va1, unsigned int va2)
```

Where:

`va1` holds the first four 8-bit summands for each addition

`va2` holds the second four 8-bit summands for each addition.

The `__uadd8` intrinsic returns:

- the addition of the first bytes in each operand, in the first byte of the return value
- the addition of the second bytes in each operand, in the second byte of the return value
- the addition of the third bytes in each operand, in the third byte of the return value
- the addition of the fourth bytes in each operand, in the fourth byte of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If `res` is the return value, then:

- if $res[7:0] \geq 0x100$ then APSR.GE[0] = 1 else 0
- if $res[15:8] \geq 0x100$ then APSR.GE[1] = 1 else 0
- if $res[23:16] \geq 0x100$ then APSR.GE[2] = 1 else 0
- if $res[31:24] \geq 0x100$ then APSR.GE[3] = 1 else 0.

Example:

```
unsigned int add_bytes(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uadd8(va1,va2); /* res[7:0] = va1[7:0] + va2[7:0]
                           res[15:8] = va1[15:8] + va2[15:8]
                           res[23:16] = va1[23:16] + va2[23:16]
                           res[31:24] = va1[31:24] + va2[31:24]
                           */
    return res;
}
```

A.40.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.41 __uasx intrinsic

This intrinsic inserts a UASX instruction into the instruction stream generated by the compiler. It enables you to exchange the two halfwords of the second operand, add the high halfwords and subtract the low halfwords.

The GE bits in the APSR are set according to the results.

```
unsigned int __uasx(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first operand for the subtraction in the low halfword, and the first operand for the addition in the high halfword

va2 holds the second operand for the subtraction in the high halfword and the second operand for the addition in the low halfword.

The __uasx intrinsic returns:

- the subtraction of the high halfword in the second operand from the low halfword in the first operand, in the low halfword of the return value
- the addition of the high halfword in the first operand and the low halfword in the second operand, in the high halfword of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If *res* is the return value, then:

- if $res[15:0] \geq 0$ then $APSR.GE[1:0] = 11$ else 00
- if $res[31:16] \geq 0x10000$ then $APSR.GE[3:2] = 11$ else 00.

Example:

```
unsigned int exchange_add_subtract(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uasx(va1,va2); /* res[15:0] = va1[15:0] - va2[31:16]
                          res[31:16] = va1[31:16] + va2[15:0]
                          */
    return res;
}
```

A.41.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.42 __uhadd16 intrinsic

This intrinsic inserts a UHADD16 instruction into the instruction stream generated by the compiler. It enables you to perform two unsigned 16-bit integer additions, halving the results.

```
unsigned int __uhadd16(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first two 16-bit summands

va2 holds the second two 16-bit summands.

The __uhadd16 intrinsic returns:

- the halved addition of the low halfwords in each operand, in the low halfword of the return value
- the halved addition of the high halfwords in each operand, in the high halfword of the return value.

Example:

```
unsigned int add_halfwords_then_halfve(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uhadd16(va1,va2); /* res[15:0] = (va1[15:0] + va2[15:0]) >> 1
                             res[31:16] = (va1[31:16] + va2[31:16]) >> 1
                             */
    return res;
}
```

A.42.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.43 `__uhadd8` intrinsic

This intrinsic inserts a UHADD8 instruction into the instruction stream generated by the compiler. It enables you to perform four unsigned 8-bit integer additions, halving the results.

```
unsigned int __uhadd8(unsigned int va1, unsigned int va2)
```

Where:

`va1` holds the first four 8-bit summands

`va2` holds the second four 8-bit summands.

The `__uhadd8` intrinsic returns:

- the halved addition of the first bytes in each operand, in the first byte of the return value
- the halved addition of the second bytes in each operand, in the second byte of the return value
- the halved addition of the third bytes in each operand, in the third byte of the return value
- the halved addition of the fourth bytes in each operand, in the fourth byte of the return value.

Example:

```
unsigned int add_bytes_then_halve(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uhadd8(va1,va2); /* res[7:0] = (va1[7:0] + va2[7:0]) >> 1
                             res[15:8] = (va1[15:8] + va2[15:8]) >> 1
                             res[23:16] = (va1[23:16] + va2[23:16]) >> 1
                             res[31:24] = (va1[31:24] + va2[31:24]) >> 1
                             */
    return res;
}
```

A.43.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.44 __uhasx intrinsic

This intrinsic inserts a UHASX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of the second operand, add the high halfwords and subtract the low halfwords, halving the results.

```
unsigned int __uhasx(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first operand for the subtraction in the low halfword, and the first operand for the addition in the high halfword

va2 holds the second operand for the subtraction in the high halfword, and the second operand for the addition in the low halfword.

The __uhasx intrinsic returns:

- the halved subtraction of the high halfword in the second operand from the low halfword in the first operand
- the halved addition of the high halfword in the first operand and the low halfword in the second operand.

Example:

```
unsigned int exchange_add_subtract(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uhasx(va1,va2); /* res[15:0] = (va1[15:0] - va2[31:16]) >> 1
                           res[31:16] = (va1[31:16] + va2[15:0]) >> 1
                           */
    return res;
}
```

A.44.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.45 __uhsax intrinsic

This intrinsic inserts a UHSAX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of the second operand, subtract the high halfwords and add the low halfwords, halving the results.

```
unsigned int __uhsax(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first operand for the addition in the low halfword, and the first operand for the subtraction in the high halfword

va2 holds the second operand for the addition in the high halfword, and the second operand for the subtraction in the low halfword.

The __uhsax intrinsic returns:

- the halved addition of the high halfword in the second operand and the low halfword in the first operand, in the low halfword of the return value
- the halved subtraction of the low halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

Example:

```
unsigned int exchange_subtract_add(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uhsax(va1,va2); /* res[15:0] = (va1[15:0] + va2[31:16]) >> 1
                           res[31:16] = (va1[31:16] - va2[15:0]) >> 1
                           */
    return res;
}
```

A.45.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.46 __uhsb16 intrinsic

This intrinsic inserts a UHSUB16 instruction into the instruction stream generated by the compiler. It enables you to perform two unsigned 16-bit integer subtractions, halving the results.

```
unsigned int __uhsb16(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first two 16-bit operands
va2 holds the second two 16-bit operands.

The __uhsb16 intrinsic returns:

- the halved subtraction of the low halfword in the second operand from the low halfword in the first operand, in the low halfword of the return value
- the halved subtraction of the high halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

Example:

```
unsigned int subtract_and_halve(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uhsb16(va1,va2); /* res[15:0] = (va1[15:0] + va2[15:0]) >> 1
                           res[31:16] = (va1[31:16] - va2[31:16]) >> 1
                           */
    return res;
}
```

A.46.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.47 __uhsb8 intrinsic

This intrinsic inserts a UHSUB8 instruction into the instruction stream generated by the compiler. It enables you to perform four unsigned 8-bit integer subtractions, halving the results.

```
unsigned int __uhsb8(unsigned int va11, unsigned int va12)
```

Where:

va11 holds the first four 8-bit operands

va12 holds the second four 8-bit operands.

The __uhsb8 intrinsic returns:

- the halved subtraction of the first byte in the second operand from the first byte in the first operand, in the first byte of the return value
- the halved subtraction of the second byte in the second operand from the second byte in the first operand, in the second byte of the return value
- the halved subtraction of the third byte in the second operand from the third byte in the first operand, in the third byte of the return value
- the halved subtraction of the fourth byte in the second operand from the fourth byte in the first operand, in the fourth byte of the return value.

Example:

```
unsigned int subtract_and_halve(unsigned int va11, unsigned int va12)
{
    unsigned int res;

    res = __uhsb8(va11,va12); /* res[7:0] = (va11[7:0] - va12[7:0]) >> 1
                               res[15:8] = (va11[15:8] - va12[15:8]) >> 1
                               res[23:16] = (va11[23:16] - va12[23:16]) >> 1
                               res[31:24] = (va11[31:24] - va12[31:24]) >> 1
                               */
    return res;
}
```

A.47.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.48 __uqadd16 intrinsic

This intrinsic inserts a UQADD16 instruction into the instruction stream generated by the compiler. It enables you to perform two unsigned 16-bit integer additions, saturating the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$.

```
unsigned int __uqadd16(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first two halfword summands
va2 holds the second two halfword summands.

The __uqadd16 intrinsic returns:

- the addition of the low halfword in the first operand and the low halfword in the second operand
- the addition of the high halfword in the first operand and the high halfword in the second operand, in the high halfword of the return value.

The results are saturated to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$.

Example:

```
unsigned int add_halfwords(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uqadd16(va1,va2); /* res[15:0] = va1[15:0] + va2[15:0]
                             res[31:16] = va1[31:16] + va2[31:16]
                             */
    return res;
}
```

A.48.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.49 __uqadd8 intrinsic

This intrinsic inserts a UQADD8 instruction into the instruction stream generated by the compiler. It enables you to perform four unsigned 8-bit integer additions, saturating the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$.

```
unsigned int __uqadd8(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first four 8-bit summands

va2 holds the second four 8-bit summands.

The __uqadd8 intrinsic returns:

- the addition of the first bytes in each operand, in the first byte of the return value
- the addition of the second bytes in each operand, in the second byte of the return value
- the addition of the third bytes in each operand, in the third byte of the return value
- the addition of the fourth bytes in each operand, in the fourth byte of the return value.

The results are saturated to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$.

Example:

```
unsigned int add_bytes(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uqadd8(va1,va2); /* res[7:0] = va1[7:0] + va2[7:0]
                             res[15:8] = va1[15:8] + va2[15:8]
                             res[23:16] = va1[23:16] + va2[23:16]
                             res[31:24] = va1[31:24] + va2[31:24]
                             */

    return res;
}
```

A.49.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.50 __uqasx intrinsic

This intrinsic inserts a UQASX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of the second operand and perform one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturating the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$.

```
unsigned int __uqasx(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first two halfword operands

va2 holds the second two halfword operands.

The __uqasx intrinsic returns:

- the subtraction of the high halfword in the second operand from the low halfword in the first operand, in the low halfword of the return value
- the subtraction of the low halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

The results are saturated to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$.

Example:

```
unsigned int exchange_add_subtract(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uqasx(va1,va2); /* res[15:0] = va1[15:0] - va2[31:16]
                           res[31:16] = va1[31:16] + va2[15:0]
                           */

    return res;
}
```

A.50.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.51 __uqsax intrinsic

This intrinsic inserts a UQSAX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of the second operand and perform one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturating the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$.

```
unsigned int __uqsax(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first 16-bit operand for the addition in the low halfword, and the first 16-bit operand for the subtraction in the high halfword

va2 holds the second 16-bit halfword for the addition in the high halfword, and the second 16-bit halfword for the subtraction in the low halfword.

The __uqsax intrinsic returns:

- the addition of the low halfword in the first operand and the high halfword in the second operand, in the low halfword of the return value
- the subtraction of the low halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

The results are saturated to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$.

Example:

```
unsigned int exchange_subtract_add(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uqsax(va1,va2); /* res[15:0] = va1[15:0] + va2[31:16]
                           res[31:16] = va1[31:16] - va2[15:0]
                           */
    return res;
}
```

A.51.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.52 __uqsub16 intrinsic

This intrinsic inserts a UQSUB16 instruction into the instruction stream generated by the compiler. It enables you to perform two unsigned 16-bit integer subtractions, saturating the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$.

```
unsigned int __uqsub16(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first halfword operands for each subtraction
va2 holds the second halfword operands for each subtraction.

The __uqsub16 intrinsic returns:

- the subtraction of the low halfword in the second operand from the low halfword in the first operand, in the low halfword of the return value
- the subtraction of the high halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

The results are saturated to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$.

Example:

```
unsigned int subtract_halfwords(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uqsub16(va1,va2); /* res[15:0] = va1[15:0] - va2[15:0]
                             *          res[31:16] = va1[31:16] - va2[31:16]
                             */

    return res;
}
```

A.52.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.53 __uqsub8 intrinsic

This intrinsic inserts a UQSUB8 instruction into the instruction stream generated by the compiler. It enables you to perform four unsigned 8-bit integer subtractions, saturating the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$.

```
unsigned int __uqsub8(unsigned int va11, unsigned int va12)
```

Where:

va11 holds the first four 8-bit operands

va12 holds the second four 8-bit operands.

The __uqsub8 intrinsic returns:

- the subtraction of the first byte in the second operand from the first byte in the first operand, in the first byte of the return value
- the subtraction of the second byte in the second operand from the second byte in the first operand, in the second byte of the return value
- the subtraction of the third byte in the second operand from the third byte in the first operand, in the third byte of the return value
- the subtraction of the fourth byte in the second operand from the fourth byte in the first operand, in the fourth byte of the return value.

The results are saturated to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$.

Example:

```
unsigned int subtract_bytes(unsigned int va11, unsigned int va12)
{
    unsigned int res;

    res = __uqsub8(va11,va12); /* res[7:0] = va11[7:0] - va12[7:0]
                               res[15:8] = va11[15:8] - va12[15:8]
                               res[23:16] = va11[23:16] - va12[23:16]
                               res[31:24] = va11[31:24] - va12[31:24]
                               */
    return res;
}
```

A.53.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [Parallel add and subtract on page 3-21 in the Assembler Reference.](#)

A.54 __usad8 intrinsic

This intrinsic inserts a USAD8 instruction into the instruction stream generated by the compiler. It enables you to perform four unsigned 8-bit subtractions, and add the absolute values of the differences together, returning the result as a single unsigned integer.

```
unsigned int __usad8(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first four 8-bit operands for the subtractions
va2 holds the second four 8-bit operands for the subtractions.

The __usad8 intrinsic returns the sum of the absolute differences of:

- the subtraction of the first byte in the second operand from the first byte in the first operand
- the subtraction of the second byte in the second operand from the second byte in the first operand
- the subtraction of the third byte in the second operand from the third byte in the first operand
- the subtraction of the fourth byte in the second operand from the fourth byte in the first operand.

The sum is returned as a single unsigned integer.

Example:

```
unsigned int subtract_add_abs(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __usad8(va1,va2); /* absdiff1 = va1[7:0] - va2[7:0]
                           absdiff2 = va1[15:8] - va2[15:8]
                           absdiff3 = va1[23:16] - va2[23:16]
                           absdiff4 = va1[31:24] - va2[31:24]
                           res[31:0] = absdiff1 + absdiff2 + absdiff3
                                   + absdiff4
                           */
    return res;
}
```

A.54.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [USAD8 and USADA8](#) on page 3-166 in the *Assembler Reference*.

A.55 __usada8 intrinsic

This intrinsic inserts a USADA8 instruction into the instruction stream generated by the compiler. It enables you to perform four unsigned 8-bit subtractions, and add the absolute values of the differences to a 32-bit accumulate operand.

```
unsigned int __usada8(unsigned int va11, unsigned int va12, unsigned int va13)
```

Where:

va11 holds the first four 8-bit operands for the subtractions
va12 holds the second four 8-bit operands for the subtractions
va13 holds the accumulation value.

The __usada8 intrinsic returns the sum of the absolute differences of the following bytes, added to the accumulation value:

- the subtraction of the first byte in the second operand from the first byte in the first operand
- the subtraction of the second byte in the second operand from the second byte in the first operand
- the subtraction of the third byte in the second operand from the third byte in the first operand
- the subtraction of the fourth byte in the second operand from the fourth byte in the first operand.

Example:

```
unsigned int subtract_add_diff_accumulate(unsigned int va11, unsigned int va12,
unsigned int va13)
{
    unsigned int res;

    res = __usada8(va11,va12,va13); /* absdiff1 = va11[7:0] - va12[7:0]
                                     absdiff2 = va11[15:8] - va12[15:8]
                                     absdiff3 = va11[23:16] - va12[23:16]
                                     absdiff4 = va11[31:24] - va12[31:24]
                                     sum = absdiff1 + absdiff2 + absdiff3
                                       + absdiff4
                                     res[31:0] = sum[31:0] + va13[31:0]
                                     */
    return res;
}
```

A.55.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [USAD8 and USADA8 on page 3-166 in the Assembler Reference.](#)

A.56 __usax intrinsic

This intrinsic inserts a USAX instruction into the instruction stream generated by the compiler. It enables you to exchange the halfwords of the second operand, subtract the high halfwords and add the low halfwords.

The GE bits in the APSR are set according to the results.

```
unsigned int __usax(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first operand for the addition in the low halfword, and the first operand for the subtraction in the high halfword

va2 holds the second operand for the addition in the high halfword, and the second operand for the subtraction in the low halfword.

The __usax intrinsic returns:

- the addition of the low halfword in the first operand and the high halfword in the second operand, in the low halfword of the return value
- the subtraction of the low halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If *res* is the return value, then:

- if $res[15:0] \geq 0x10000$ then $APSR.GE[1:0] = 11$ else 00
- if $res[31:16] \geq 0$ then $APSR.GE[3:2] = 11$ else 00.

Example:

```
unsigned int exchange_subtract_add(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __usax(va1,va2); /* res[15:0] = va1[15:0] + va2[31:16]
                          res[31:16] = va1[31:16] - va2[15:0]
                          */
    return res;
}
```

A.56.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.57 __usat16 intrinsic

This intrinsic inserts a USAT16 instruction into the instruction stream generated by the compiler. It enables you to saturate two signed 16-bit values to a selected unsigned range. The Q flag is set if either operation saturates.

```
unsigned int __usat16(unsigned int va11, /* constant */ unsigned int va12)
```

Where:

va11 holds the two 16-bit values that are to be saturated

va12 specifies the bit position for saturation, and must be an integral constant expression.

The __usat16 intrinsic returns the saturation of the two signed 16-bit values, as non-negative values.

Example:

```
unsigned int saturate_halfwords(unsigned int va11, unsigned int va12)
{
    unsigned int res;

    res = __usax(va11,va12); /* Saturate halfwords in va11 to the unsigned
                             range specified by the bit position in va12
                             */    return res;
}
```

A.57.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [SSAT16 and USAT16 on page 3-150 in the Assembler Reference.](#)

A.58 __usub16 intrinsic

This intrinsic inserts a USUB16 instruction into the instruction stream generated by the compiler. It enables you to perform two 16-bit unsigned integer subtractions.

The GE bits in the APSR are set according to the results.

```
unsigned int __usub16(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first two halfword operands

va2 holds the second two halfword operands.

The __usub16 intrinsic returns:

- the subtraction of the low halfword in the second operand from the low halfword in the first operand, in the low halfword of the return value
- the subtraction of the high halfword in the second operand from the high halfword in the first operand, in the high halfword of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If *res* is the return value, then:

- if $res[15:0] \geq 0$ then $APSR.GE[1:0] = 11$ else 00
- if $res[31:16] \geq 0$ then $APSR.GE[3:2] = 11$ else 00.

Example:

```
unsigned int subtract_halfwords(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __usub16(va1,va2); /* res[15:0] = va1[15:0] - va2[15:0]
                           *      res[31:16] = va1[31:16] - va2[31:16]
                           */
}
```

A.58.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.59 __usub8 intrinsic

This intrinsic inserts a USUB8 instruction into the instruction stream generated by the compiler. It enables you to perform four 8-bit unsigned integer subtractions.

The GE bits in the APSR are set according to the results.

```
unsigned int __usub8(unsigned int va1, unsigned int va2)
```

Where:

va1 holds the first four 8-bit operands

va2 holds the second four 8-bit operands.

The __usub8 intrinsic returns:

- the subtraction of the first byte in the second operand from the first byte in the first operand, in the first byte of the return value
- the subtraction of the second byte in the second operand from the second byte in the first operand, in the second byte of the return value
- the subtraction of the third byte in the second operand from the third byte in the first operand, in the third byte of the return value
- the subtraction of the fourth byte in the second operand from the fourth byte in the first operand, in the fourth byte of the return value.

Each bit in APSR.GE is set or cleared for each byte in the return value, depending on the results of the operation. If *res* is the return value, then:

- if $res[7:0] \geq 0$ then APSR.GE[0] = 1 else 0
- if $res[15:8] \geq 0$ then APSR.GE[1] = 1 else 0
- if $res[23:16] \geq 0$ then APSR.GE[2] = 1 else 0
- if $res[31:24] \geq 0$ then APSR.GE[3] = 1 else 0.

Example:

```
unsigned int subtract(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __usub8(va1,va2); /* res[7:0] = va1[7:0] - va2[7:0]
                           res[15:8] = va1[15:8] - va2[15:8]
                           res[23:16] = va1[23:16] - va2[23:16]
                           res[31:24] = va1[31:24] - va2[31:24]
                           */
}
```

A.59.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [Parallel add and subtract](#) on page 3-21 in the *Assembler Reference*.

A.60 __uxtab16 intrinsic

This intrinsic inserts a UXTAB16 instruction into the instruction stream generated by the compiler. It enables you to extract two 8-bit values from one operand, zero-extend them to 16 bits each, and add the results to two 16-bit values from another operand.

```
unsigned int __uxtab16(unsigned int va1, unsigned int va2)
```

Where `va2[7:0]` and `va2[23:16]` hold the two 8-bit values to be zero-extended.

The `__uxtab16` intrinsic returns the 8-bit values in `va2`, zero-extended to 16-bit values and added to `va1`.

Example:

```
unsigned int extend_add(unsigned int va1, unsigned int va2)
{
    unsigned int res;

    res = __uxtab16(va1,va2); /* res[15:0] = ZeroExt(va2[7:0] to 16 bits)
                             + va1[15:0]
                             res[31:16] = ZeroExt(va2[31:16] to 16 bits)
                             + va1[31:16]
                             */
    return res;
}
```

A.60.1 See also

- [ARMv6 SIMD intrinsics](#) on page 5-168
- [Instruction summary](#) on page 3-2 in the *Assembler Reference*
- [SXT, SXTA, UXT, and UXTA](#) on page 3-156 in the *Assembler Reference*.

A.61 __uxtb16 intrinsic

This intrinsic inserts a UXTB16 instruction into the instruction stream generated by the compiler. It enables you to extract two 8-bit values from an operand and zero-extend them to 16 bits each.

```
unsigned int __uxtb16(unsigned int va1)
```

Where `va1[7:0]` and `va1[23:16]` hold the two 8-bit values to be zero-extended.

The `__uxtb16` intrinsic returns the 8-bit values zero-extended to 16-bit values.

Example:

```
unsigned int zero_extend(unsigned int va1)
{
    unsigned int res;

    res = __uxtb16(va1,va1); /* res[15:0] = ZeroExtended(va1[7:0])
                             res[31:16] = ZeroExtended(va1[23:16])
                             */
    return res;
}
```

A.61.1 See also

- [ARMv6 SIMD intrinsics on page 5-168](#)
- [Instruction summary on page 3-2 in the Assembler Reference](#)
- [SXT, SXTA, UXT, and UXTA on page 3-156 in the Assembler Reference.](#)

Appendix B

Via File Syntax

This appendix describes the syntax of via files accepted by all the ARM development tools. It contains the following topics:

- [Overview of via files on page B-2](#)
- [Via file syntax on page B-3.](#)

B.1 Overview of via files

Via files are plain text files that contain command-line arguments and options to ARM development tools. You can use via files with all the ARM command-line tools, that is, you can specify a via file from the command line using the `--via` command-line option with:

- `armcc`
- `armasm`
- `armlink`
- `fromelf`
- `armar.`

See the documentation for the individual tool for more information.

Note

In general, you can use a via file to specify any command-line option to a tool, including `--via`. This means that you can call multiple nested via files from within a via file.

B.1.1 Via file evaluation

When a tool that supports via files is invoked it:

1. Replaces the first specified `--via via_file` argument with the sequence of argument words extracted from the via file, including recursively processing any nested `--via` commands in the via file.
2. Processes any subsequent `--via via_file` arguments in the same way, in the order they are presented.

That is, via files are processed in the order you specify them, and each via file is processed completely including processing nested via files before processing the next via file.

B.2 Via file syntax

Via files must conform to the following syntax rules:

- A via file is a text file containing a sequence of words. Each word in the text file is converted into an argument string and passed to the tool.
- Words are separated by whitespace, or the end of a line, except in delimited strings. For example:

```
--c90 --strict (two words)
--c90--strict (one word)
```
- The end of a line is treated as whitespace. For example:

```
--c90
--strict
```

is equivalent to:

```
--c90 --strict
```
- Strings enclosed in quotation marks ("), or apostrophes (') are treated as a single word. Within a quoted word, an apostrophe is treated as an ordinary character. Within an apostrophe delimited word, a quotation mark is treated as an ordinary character. Quotation marks are used to delimit filenames or path names that contain spaces. For example:

```
-I C:\My Project\includes (three words) -I "C:\My Project\includes" (two words)
```

Apostrophes can be used to delimit words that contain quotes. For example:

```
-DNAME="'RealView Compilation Tools'" (one word)
```
- Characters enclosed in parentheses are treated as a single word. For example:

```
--option(x, y, z) (one word)
--option (x, y, z) (two words)
```
- Within quoted or apostrophe delimited strings, you can use a backslash (\) character to escape the quote, apostrophe, and backslash characters.
- A word that occurs immediately next to a delimited word is treated as a single word. For example:

```
-I"C:\Project\includes"
```

is treated as the single word:

```
-IC:\Project\includes
```
- Lines beginning with a semicolon (;) or a hash (#) character as the first nonwhitespace character are comment lines. If a semicolon or hash character appears anywhere else in a line, it is not treated as the start of a comment. For example:

```
-o objectname.axf ;this is not a comment
```

A comment ends at the end of a line, or at the end of the file. There are no multi-line comments, and there are no part-line comments.
- Lines that include the preprocessor option `-Dsymbol="value"` must be delimited with a single quote, either as `'-Dsymbol=value'` or as `-Dsymbol='value'`. For example:

```
-c -DFOO_VALUE='FOO_VALUE'
```

Appendix C

Summary Table of GNU Language Extensions

GNU provides many extensions to the C and C++ languages. These extensions are also supported by the ARM compiler in GNU mode (for example GNU C90). Some extensions are supported in a non GNU mode (for example C90). This appendix lists the language extensions that the ARM compiler supports and the modes they are supported in. The Origin column shows whether the language feature is part of any of the C90, C99, or C++ ISO Standards. The Origin column shows GCC-specific if the feature originated as a GCC extension.

Table C-1 Supported GNU extensions

GNU extension	Origin	Modes supported
__alignof__ on page 5-7	GCC-Specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
Aggregate initializer elements for automatic variables	Standard C99, Standard C++.	C99, C++, GNU C90, GNU C99, GNU C++.
Alternate keywords	GCC-specific.	GNU C90, GNU C99, GNU C++.
<code>asm</code> keyword	Standard C++.	C++, GNU C90, GNU C++.
Assembler labels	-	C90, C99, C++, GNU C90, GNU C99, GNU C++.
Case ranges	GCC-specific.	GNU C90, GNU C99, GNU C++.
Cast of a union	GCC-specific.	GNU C90, GNU C99.

Table C-1 Supported GNU extensions (continued)

GNU extension	Origin	Modes supported
Character escape sequence	GCC-specific.	GNU C90, GNU C99, GNU C++.
Compound literals	Standard C99.	C99, GNU C90, GNU C99, GNU C++.
Conditional statements with omitted operands	GCC-specific.	GNU C90, GNU C99, GNU C++.
Designated initializers	Standard C99.	C99, GNU C90, GNU C99, GNU C++.
Dollar signs in identifiers	GCC-specific.	GNU C90, GNU C99, GNU C++.
Extended lvalues	Standard C++.	C++, GNU C90, GNU C99, GNU C++. ^a
<i>Function attributes on page 5-39</i>	-	C90, C99, C++, GNU C90, GNU C99, GNU C++.
<i>GNU built-in functions on page 5-180</i>	-	-
Inline functions	Standard C99, Standard C++.	C99, C++, GNU C90, GNU C99, GNU C++.
Labels as values	GCC-specific.	GNU C90, GNU C99, GNU C++.
Pointer arithmetic on void pointers and function pointers	GCC-specific.	GNU C90, GNU C99.
Statement expressions	GCC-specific.	GNU C90, GNU C99, GNU C++.
Unnamed embedded structures or unions	GCC-specific.	GNU C90, GNU C99, GNU C++.
<i>__attribute__((aligned)) variable attribute on page 5-73</i>	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
<i>__attribute__((deprecated)) variable attribute on page 5-74</i>	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
<i>__attribute__((packed)) variable attribute on page 5-76</i>	GCC-specific.	C90, C99, GNU C90, GNU C99, GNU C++.
<i>__attribute__((section("name"))) variable attribute on page 5-77</i>	GCC-specific.	C99, GNU C90, GNU C99, GNU C++.
<i>__attribute__((transparent_union)) variable attribute on page 5-78</i>	GCC-specific.	GNU C90, GNU C99.
<i>__attribute__((unused)) variable attribute on page 5-79</i>	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
<i>__attribute__((used)) variable attribute on page 5-80</i>	GCC-specific.	C90, C99, GNU C90, GNU C99.

Table C-1 Supported GNU extensions (continued)

GNU extension	Origin	Modes supported
__attribute__((weak)) variable attribute on page 5-82	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
Variadic macros	Standard C99.	C90, C99, C++, GNU C90, GNU C99, GNU C++ ^b .
Zero-length arrays	GCC-specific.	GNU C90, GNU C99.

a. Only accepted for certain values of `--gnu_version`.

b. If `--gnu` is specified (GNU modes), GNU-specific syntax applies.

Other information

- *Which GNU language extensions are supported by the ARM Compiler?*, <http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka14717.html>.

Appendix D

Standard C Implementation Definition

This appendix gives information required by the ISO C standard for conforming C implementations. It contains the following topics:

- *Implementation definition* on page D-2
 - *Translation* on page D-3
 - *Environment* on page D-4
 - *Identifiers* on page D-6
 - *Characters* on page D-7
 - *Integers* on page D-8
 - *Floating-point* on page D-9
 - *Arrays and pointers* on page D-10
 - *Registers* on page D-11
 - *Structures, unions, enumerations, and bitfields* on page D-12
 - *Qualifiers* on page D-13
 - *Expression evaluation* on page D-14
 - *Preprocessing directives* on page D-15
 - *Library functions* on page D-16
- *Behaviors considered undefined by the ISO C Standard* on page D-17.

D.1 Implementation definition

Appendix G of the ISO C standard (ISO/IEC 9899:1990 (E)) collates information about portability issues. Sub-clause G3 lists the behavior that each implementation must document.

———— **Note** —————

This appendix does not duplicate information that is part of [Chapter 5 Compiler-specific Features](#). This appendix provides references where applicable.

The following topics correspond to the relevant sections of sub-clause G3. They describe aspects of the ARM C compiler and C library, not defined by the ISO C standard, that are implementation-defined:

- [Translation](#) on page D-3
- [Environment](#) on page D-4
- [Identifiers](#) on page D-6
- [Characters](#) on page D-7
- [Integers](#) on page D-8
- [Floating-point](#) on page D-9
- [Arrays and pointers](#) on page D-10
- [Registers](#) on page D-11
- [Structures, unions, enumerations, and bitfields](#) on page D-12
- [Qualifiers](#) on page D-13
- [Expression evaluation](#) on page D-14
- [Preprocessing directives](#) on page D-15
- [Library functions](#) on page D-16.

———— **Note** —————

The support for the `wctype.h` and `wchar.h` headers excludes wide file operations.

D.2 Translation

Diagnostic messages produced by the compiler are of the form:

source-file, *line-number*: *severity*: *error-code*: *explanation*

where *severity* is one of:

- [blank] If the severity is blank, this is a remark and indicates common, but sometimes unconventional, use of C or C++. Remarks are not displayed by default. Use the `--remarks` option to display remark messages. Compilation continues.
- Warning Flags unusual conditions in your code that might indicate a problem. Compilation continues.
- Error Indicates a problem that causes the compilation to stop. For example, violations in the syntactic or semantic rules of the C or C++ language.

Internal fault

Indicates an internal problem with the compiler. Contact your supplier with the information listed in [Chapter 1 Conventions and Feedback](#).

Here:

error-code Is a number identifying the error type.

explanation Is a text description of the error.

See [Chapter 7 Compiler Diagnostic Messages](#) in *Using the Compiler* for more information.

D.3 Environment

The mapping of a command line from the ARM architecture-based environment into arguments to `main()` is implementation-specific. The generic ARM C library supports the following:

D.3.1 `main()`

The arguments given to `main()` are the words of the command line not including input/output redirections, delimited by whitespace, except where the whitespace is contained in double quotes.

———— **Note** —————

- A whitespace character is any character where the result of `isspace()` is true.
- A double quote or backslash character `\` inside double quotes must be preceded by a backslash character.
- An input/output redirection is not recognized inside double quotes.

D.3.2 Interactive device

In a nonhosted implementation of the ARM C library, the term *interactive device* might be meaningless. The generic ARM C library supports a pair of devices, both called `:tt`, intended to handle keyboard input and VDU screen output. In the generic implementation:

- no buffering is done on any stream connected to `:tt` unless input/output redirection has occurred
- if input/output redirection other than to `:tt` has occurred, full file buffering is used except that line buffering is used if both `stdout` and `stderr` were redirected to the same file.

D.3.3 Redirecting standard input, output, and error streams

Using the generic ARM C library, the standard input, output and error streams can be redirected at runtime. For example, if `mycopy` is a program running on a host debugger that copies the standard input to the standard output, the following line runs the program:

```
mycopy < infile > outfile 2> errfile
```

and redirects the files as follows:

```
stdin      The standard input stream is redirected to infile.
stdout     The standard output stream is redirected to outfile.
stderr     The standard error stream is redirected to errfile.
```

The permitted redirections are:

```
0< filename Reads stdin from filename.
< filename  Reads stdin from filename.
1> filename Writes stdout to filename.
> filename  Writes stdout to filename.
2> filename Writes stderr to filename.
2>&1        Writes stderr to the same place as stdout.
```

>& *file* Writes both stdout and stderr to *filename*.

>> *filename* Appends stdout to *filename*.

>>& *filename* Appends both stdout and stderr to *filename*.

To redirect stdin, stdout, and stderr on the target, you must define:

```
#pragma import(_main_redirection)
```

File redirection is done only if either:

- the invoking operating system supports it
- the program reads and writes characters and has not replaced the C library functions `fputc()` and `fgetc()`.

D.4 Identifiers

See [Character sets and identifiers on page 6-3](#) for more information.

D.5 Characters

See [Character sets and identifiers on page 6-3](#) for more information.

D.6 Integers

See [Integer](#) on page 6-5 for more information.

D.7 Floating-point

See [Float](#) on page 6-5 for more information.

D.8 Arrays and pointers

See [Arrays and pointers](#) on page 6-6 for more information.

D.9 Registers

Using the ARM compiler, you can declare any number of local objects to have the storage class **register**.

D.10 Structures, unions, enumerations, and bitfields

The ISO/IEC C standard requires the following implementation details to be documented for structured data types:

- the outcome when a member of a union is accessed using a member of different type
- the padding and alignment of members of structures
- whether a plain `int` bitfield is treated as a **signed int** bitfield or as an **unsigned int** bitfield
- the order of allocation of bitfields within a unit
- whether a bitfield can straddle a storage-unit boundary
- the integer type chosen to represent the values of an enumeration type.

See [Chapter 6 C and C++ Implementation Details](#) for more information.

D.10.1 Unions

See [Unions on page 6-9](#) for information.

D.10.2 Enumerations

See [Enumerations on page 6-9](#) for information.

D.10.3 Padding and alignment of structures

See [Structures on page 6-10](#) for information.

D.10.4 Bitfields

See [Bitfields on page 6-11](#) for information.

D.11 Qualifiers

An object that has a volatile-qualified type is accessed as a word, halfword, or byte as determined by its size and alignment. For volatile objects larger than a word, the order of accesses to the parts of the object is undefined. Updates to volatile bitfields generally require a read-modify-write. Accesses to aligned word, halfword and byte types are atomic. Other volatile accesses are not necessarily atomic.

Otherwise, reads and writes to volatile qualified objects occur as directly implied by the source code, in the order implied by the source code.

D.12 Expression evaluation

The compiler can re-order expressions involving only associative and commutative operators of equal precedence, even in the presence of parentheses. For example, $a + (b + c)$ might be evaluated as $(a + b) + c$ if a , b , and c are integer expressions.

Between sequence points, the compiler can evaluate expressions in any order, regardless of parentheses. Therefore, side effects of expressions between sequence points can occur in any order.

The compiler can evaluate function arguments in any order.

Any aspect of evaluation order not prescribed by the relevant standard can be varied by:

- the optimization level you are compiling at
- the release of the compiler you are using.

D.13 Preprocessing directives

The ISO standard C header files can be referred to as described in the standard, for example, `#include <stdio.h>`.

Quoted names for includable source files are supported. The compiler accepts host filenames or UNIX filenames. For UNIX filenames on non-UNIX hosts, the compiler tries to translate the filename to a local equivalent.

The following C99 pragmas are recognized by the compiler, but ignored:

STDC CX_LIMITED_RANGE

See *ISO/IEC 9899:1999/Cor 2:2004*, Section 7.3.4.

STDC FENV_ACCESS

See *ISO/IEC 9899:1999/Cor 2:2004*, Section 7.6.1.

STDC FP_CONTRACT

See *ISO/IEC 9899:1999/Cor 2:2004*, Section 7.12.2.

The supported `#pragma` directives are shown in [Pragmas on page 5-85](#).

D.13.1 See also

- [ISO C99 on page 2-5](#).

D.14 Library functions

The ISO C library variants are listed in *C and C++ runtime libraries on page 2-6* in *Using ARM C and C++ Libraries and Floating-Point Support*.

The precise nature of each C library is unique to the particular implementation. The generic ARM C library has, or supports, the following features:

- The macro NULL expands to the integer constant 0.
- If a program redefines a reserved external identifier such as printf, an error might occur when the program is linked with the standard libraries. If it is not linked with standard libraries, no error is detected.
- The __aeabi_assert() function prints details of the failing diagnostic on stderr and then calls the abort() function:

```
*** assertion failed: expression, file name, line number
```

———— **Note** —————

The behavior of the assert macro depends on the conditions in operation at the most recent occurrence of #include <assert.h>. See *Program exit and the assert macro on page 2-62* in *Using ARM C and C++ Libraries and Floating-Point Support* for more information.

For implementation details of mathematical functions, macros, locale, signals, and input/output see *Chapter 2 The ARM C and C++ libraries* in *Using ARM C and C++ Libraries and Floating-Point Support*.

D.15 Behaviors considered undefined by the ISO C Standard

The following are considered undefined behavior by the ISO C Standard:

- In character and string escapes, if the character following the \ has no special meaning, the value of the escape is the character itself. For example, a warning is generated if you use \s because it is the same as s.
- A **struct** that has no named fields but at least one unnamed field is accepted by default, but generates an error in strict 1990 ISO Standard C.

Appendix E

Standard C++ Implementation Definition

The ARM compiler supports the majority of the language features described in the ISO/IEC standard for C++ when compiling C++. This appendix lists the C++ language features defined in the standard, and states whether or not that language feature is supported by ARM C++:

- [Integral conversion on page E-2](#)
- [Calling a pure virtual function on page E-3](#)
- [Major features of language support on page E-4](#)
- [Standard C++ library implementation definition on page E-5.](#)

Note

This appendix does not duplicate information that is part of the standard C implementation. See [Appendix D Standard C Implementation Definition](#).

When compiling C++ in ISO C mode, the ARM compiler is identical to the ARM C compiler. Where there is an implementation feature specific to either C or C++, this is noted in the text. For extensions to Standard C++, see:

- [Standard C++ language extensions on page 4-26](#)
- [C99 language features available in C++ and C90 on page 4-12](#)
- [Standard C and Standard C++ language extensions on page 4-35.](#)

E.1 Integral conversion

During integral conversion, if the destination type is signed, the value is unchanged if it can be represented in the destination type and bitfield width. Otherwise, the value is truncated to fit the size of the destination type.

———— **Note** —————

This topic is related to Section 4.7 Integral conversions, in the ISO/IEC standard.

E.2 Calling a pure virtual function

Calling a pure virtual function is illegal. If your code calls a pure virtual function, then the compiler includes a call to the library function `__cxa_pure_virtual`.

`__cxa_pure_virtual` raises the signal **SIGPVFN**. The default signal handler prints an error message and exits. See [__default_signal_handler\(\)](#) on page 2-8 in the *ARM C and C++ Libraries and Floating-Point Support Reference* for more information.

E.3 Major features of language support

Table E-1 shows the major features of the language supported by this release of ARM C++.

Table E-1 Major feature support for language

Major feature	ISO/IEC standard section	Support
Core language	1 to 13	Yes.
Templates	14	Yes, with the exception of export templates.
Exceptions	15	Yes.
Libraries	17 to 27	See the <i>Standard C++ library implementation definition on page E-5</i> , the <i>ARM C and C++ Libraries and Floating-Point Support Reference</i> , and <i>Using ARM C and C++ Libraries and Floating-Point Support</i> .

E.4 Standard C++ library implementation definition

Version 2.02.03 of the Rogue Wave library provides a subset of the library defined in the standard. There are small differences from the 1999 ISO C standard. For information on the implementation definition, see *Standard C++ library implementation definition on page 2-114* in *Using ARM C and C++ Libraries and Floating-Point Support*.

The library can be used with user-defined functions to produce target-dependent applications. See *C and C++ runtime libraries on page 2-6* in *Using ARM C and C++ Libraries and Floating-Point Support*.

Appendix F

C and C++ Compiler Implementation Limits

This appendix lists the implementation limits when using the ARM compiler to compile C and C++. It contains the following topics:

- [C++ ISO/IEC standard limits](#) on page F-2
- [Limits for integral numbers](#) on page F-4
- [Limits for floating-point numbers](#) on page F-5.

F.1 C++ ISO/IEC standard limits

The ISO/IEC C++ standard recommends minimum limits that a conforming compiler must accept. You must be aware of these when porting applications between compilers. Table F-1 gives a summary of these limits.

In this table, a limit of memory indicates that the ARM compiler imposes no limit, other than that imposed by the available memory.

Table F-1 Implementation limits

Description	Recommended	ARM
Nesting levels of compound statements, iteration control structures, and selection control structures.	256	memory
Nesting levels of conditional inclusion.	256	memory
Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration.	256	memory
Nesting levels of parenthesized expressions within a full expression.	256	memory
Number of initial characters in an internal identifier or macro name.	1024	memory
Number of initial characters in an external identifier.	1024	memory
External identifiers in one translation unit.	65536	memory
Identifiers with block scope declared in one block.	1024	memory
Macro identifiers simultaneously defined in one translation unit.	65536	memory
Parameters in one function declaration.	256	memory
Arguments in one function call.	256	memory
Parameters in one macro definition.	256	memory
Arguments in one macro invocation.	256	memory
Characters in one logical source line.	65536	memory
Characters in a character string literal or wide string literal after concatenation.	65536	memory
Size of a C or C++ object (including arrays).	262144	4294967296
Nesting levels of #include file.	256	memory
Case labels for a switch statement, excluding those for any nested switch statements.	16384	memory
Data members in a single class, structure, or union.	16384	memory
Enumeration constants in a single enumeration.	4096	memory
Levels of nested class, structure, or union definitions in a single struct declaration-list.	256	memory
Functions registered by <code>atexit()</code> .	32	33
Direct and indirect base classes.	16384	memory
Direct base classes for a single class.	1024	memory

Table F-1 Implementation limits (continued)

Description	Recommended	ARM
Members declared in a single class.	4096	memory
Final overriding virtual functions in a class, accessible or not.	16384	memory
Direct and indirect virtual bases of a class.	1024	memory
Static members of a class.	1024	memory
Friend declarations in a class.	4096	memory
Access control declarations in a class.	4096	memory
Member initializers in a constructor definition.	6144	memory
Scope qualifications of one identifier.	256	memory
Nested external specifications.	1024	memory
Template arguments in a template declaration.	1024	memory
Recursively nested template instantiations.	17	memory
Handlers per try block.	256	memory
Throw specifications on a single function declaration.	256	memory

F.2 Limits for integral numbers

Table F-2 gives the ranges for integral numbers in ARM C and C++. The Value column of the table gives the numerical value of the range endpoint. The Hex value column gives the bit pattern (in hexadecimal) that is interpreted as this value by the ARM compiler. These constants are defined in the `limits.h` include file.

When entering a constant, choose the size and sign with care. Constants are interpreted differently in decimal and hexadecimal/octal. See the appropriate C or C++ standard, or any of the recommended C and C++ textbooks for more information, as described in [Further reading on page 2-31](#) of *Introducing the ARM Compiler toolchain*.

Table F-2 Integer ranges

Constant	Meaning	Value	Hex value
CHAR_MAX	Maximum value of <code>char</code>	255	0xFF
CHAR_MIN	Minimum value of <code>char</code>	0	0x00
SCHAR_MAX	Maximum value of <code>signed char</code>	127	0x7F
SCHAR_MIN	Minimum value of <code>signed char</code>	-128	0x80
UCHAR_MAX	Maximum value of <code>unsigned char</code>	255	0xFF
SHRT_MAX	Maximum value of <code>short</code>	32767	0x7FFF
SHRT_MIN	Minimum value of <code>short</code>	-32768	0x8000
USHRT_MAX	Maximum value of <code>unsigned short</code>	65535	0xFFFF
INT_MAX	Maximum value of <code>int</code>	2147483647	0x7FFFFFFF
INT_MIN	Minimum value of <code>int</code>	-2147483648	0x80000000
LONG_MAX	Maximum value of <code>long</code>	2147483647	0x7FFFFFFF
LONG_MIN	Minimum value of <code>long</code>	-2147483648	0x80000000
ULONG_MAX	Maximum value of <code>unsigned long</code>	4294967295	0xFFFFFFFF
LLONG_MAX	Maximum value of <code>long long</code>	9.2E+18	0x7FFFFFFF FFFFFFFF
LLONG_MIN	Minimum value of <code>long long</code>	-9.2E+18	0x80000000 00000000
ULLONG_MAX	Maximum value of <code>unsigned long long</code>	1.8E+19	0xFFFFFFFF FFFFFFFF

F.3 Limits for floating-point numbers

This topic describes the characteristics of floating-point numbers.

[Table F-3](#) gives the limits for floating-point numbers. These constants are defined in the `float.h` include file.

Table F-3 Floating-point limits

Constant	Meaning	Value
FLT_MAX	Maximum value of float	3.40282347e+38F
FLT_MIN	Minimum normalized positive floating-point number value of float	1.175494351e-38F
DBL_MAX	Maximum value of double	1.79769313486231571e+308
DBL_MIN	Minimum normalized positive floating-point number value of double	2.22507385850720138e-308
LDBL_MAX	Maximum value of long double	1.79769313486231571e+308
LDBL_MIN	Minimum normalized positive floating-point number value of long double	2.22507385850720138e-308
FLT_MAX_EXP	Maximum value of base 2 exponent for type float	128
FLT_MIN_EXP	Minimum value of base 2 exponent for type float	-125
DBL_MAX_EXP	Maximum value of base 2 exponent for type double	1024
DBL_MIN_EXP	Minimum value of base 2 exponent for type double	-1021
LDBL_MAX_EXP	Maximum value of base 2 exponent for type long double	1024
LDBL_MIN_EXP	Minimum value of base 2 exponent for type long double	-1021
FLT_MAX_10_EXP	Maximum value of base 10 exponent for type float	38
FLT_MIN_10_EXP	Minimum value of base 10 exponent for type float	-37
DBL_MAX_10_EXP	Maximum value of base 10 exponent for type double	308
DBL_MIN_10_EXP	Minimum value of base 10 exponent for type double	-307
LDBL_MAX_10_EXP	Maximum value of base 10 exponent for type long double	308
LDBL_MIN_10_EXP	Minimum value of base 10 exponent for type long double	-307

[Table F-4](#) describes other characteristics of floating-point numbers. These constants are also defined in the `float.h` include file.

Table F-4 Other floating-point characteristics

Constant	Meaning	Value
FLT_RADIX	Base (radix) of the ARM floating-point number representation	2
FLT_ROUNDS	Rounding mode for floating-point numbers	(nearest) 1
FLT_DIG	Decimal digits of precision for float	6
DBL_DIG	Decimal digits of precision for double	15

Table F-4 Other floating-point characteristics (continued)

Constant	Meaning	Value
LDBL_DIG	Decimal digits of precision for long double	15
FLT_MANT_DIG	Binary digits of precision for type float	24
DBL_MANT_DIG	Binary digits of precision for type double	53
LDBL_MANT_DIG	Binary digits of precision for type long double	53
FLT_EPSILON	Smallest positive value of x that $1.0 + x \neq 1.0$ for type float	$1.19209290e-7F$
DBL_EPSILON	Smallest positive value of x that $1.0 + x \neq 1.0$ for type double	$2.2204460492503131e-16$
LDBL_EPSILON	Smallest positive value of x that $1.0 + x \neq 1.0$ for type long double	$2.2204460492503131e-16L$

Note

- When a floating-point number is converted to a shorter floating-point number, it is rounded to the nearest representable number.
 - Floating-point arithmetic conforms to IEEE 754.
-

Appendix G

Using NEON Support

This appendix describes NEON intrinsics support in this release of the ARM compilation tools. It contains the following topics:

- [Introduction](#) on page G-3
- [Vector data types](#) on page G-4
- [Summary of NEON intrinsics](#) on page G-5
- [Intrinsics](#) on page G-10
 - [Addition](#) on page G-12
 - [Multiplication](#) on page G-14
 - [Subtraction](#) on page G-16
 - [Comparison](#) on page G-18
 - [Absolute difference](#) on page G-21
 - [Max/Min](#) on page G-22
 - [Pairwise addition](#) on page G-23
 - [Folding maximum](#) on page G-24
 - [Folding minimum](#) on page G-25
 - [Reciprocal/Sqrt](#) on page G-26
 - [Shifts by signed variable](#) on page G-27
 - [Shifts by a constant](#) on page G-29
 - [Shifts with insert](#) on page G-33
 - [Loads of a single vector or lane](#) on page G-34
 - [Store a single vector or lane](#) on page G-37
 - [Loads of an N-element structure](#) on page G-39
 - [Extract lanes from a vector and put into a register](#) on page G-48

- *Load a single lane of a vector from a literal* on page G-49
- *Initialize a vector from a literal bit pattern* on page G-50
- *Set all lanes to same value* on page G-51
- *Combining vectors* on page G-53
- *Splitting vectors* on page G-54
- *Converting vectors* on page G-55
- *Table look up* on page G-57
- *Extended table look up intrinsics* on page G-58
- *Operations with a scalar value* on page G-59
- *Vector extract* on page G-64
- *Reverse vector elements (swap endianness)* on page G-65
- *Other single operand arithmetic* on page G-66
- *Logical operations* on page G-68
- *Transposition operations* on page G-71
- *Vector reinterpret cast operations* on page G-73.

G.1 Introduction

The ARM compilation tools provide intrinsics to generate NEON code for all Cortex-A series processors in both ARM and Thumb state. The NEON intrinsics are defined in the header file `arm_neon.h`. The header file defines both the intrinsics and a set of vector types.

There is no support for NEON intrinsics for architectures before ARMv7. When building for earlier architectures, or for ARMv7 architecture profiles that do not include NEON, the compiler treats NEON intrinsics as ordinary function calls. This results in an error at link time.

G.2 Vector data types

The following types are defined to represent vectors. NEON vector data types are named according to the following pattern:

<type><size>x<number of lanes>_t

For example, `int16x4_t` is a vector containing four lanes each containing a signed 16-bit integer. [Table G-1](#) lists the vector data types.

Table G-1 Vector data types

<code>int8x8_t</code>	<code>int8x16_t</code>
<code>int16x4_t</code>	<code>int16x8_t</code>
<code>int32x2_t</code>	<code>int32x4_t</code>
<code>int64x1_t</code>	<code>int64x2_t</code>
<code>uint8x8_t</code>	<code>uint8x16_t</code>
<code>uint16x4_t</code>	<code>uint16x8_t</code>
<code>uint32x2_t</code>	<code>uint32x4_t</code>
<code>uint64x1_t</code>	<code>uint64x2_t</code>
<code>float16x4_t</code>	<code>float16x8_t</code>
<code>float32x2_t</code>	<code>float32x4_t</code>
<code>poly8x8_t</code>	<code>poly8x16_t</code>
<code>poly16x4_t</code>	<code>poly16x8_t</code>

Some intrinsics use an array of vector types of the form:

<type><size>x<number of lanes>x<length of array>_t

These types are treated as ordinary C structures containing a single element named `val`.

An example structure definition is:

```
struct int16x4x2_t
{
    int16x4_t val[2];
};
```

There are array types defined for array lengths between 2 and 4, with any of the vector types listed in [Table G-1](#).

———— **Note** ————

The vector data types and arrays of the vector data types cannot be initialized by direct literal assignment. You must initialize them using one of the load intrinsics. See [Table G-2 on page G-5](#).

G.3 Summary of NEON intrinsics

This provides a summary of the NEON intrinsics categories. Use it to locate individual intrinsics. This contains:

Table G-2 Summary of NEON intrinsics

Category	Intrinsic description
Addition	<i>Vector add: vadd{q}_<type>. Vr[i]:=Va[i]+Vb[i] on page G-12</i>
	<i>Vector long add: vaddl_<type>. Vr[i]:=Va[i]+Vb[i] on page G-12</i>
	<i>Vector wide add: vaddw_<type>. Vr[i]:=Va[i]+Vb[i] on page G-12</i>
	<i>Vector halving add: vhadd{q}_<type>. Vr[i:]=(Va[i]+Vb[i])>>1 on page G-12</i>
	<i>Vector rounding halving add: vrhadd{q}_<type>. Vr[i:]=(Va[i]+Vb[i]+1)>>1 on page G-13</i>
	<i>VQADD: Vector saturating add on page G-13</i>
	<i>Vector add high half: vaddhn_<type>. Vr[i]:=Va[i]+Vb[i] on page G-13</i>
	<i>Vector rounding add high half: vraddhn_<type>. on page G-13</i>
	Multiplication
<i>Vector multiply accumulate: vmla{q}_<type>. Vr[i] := Va[i] + Vb[i] * Vc[i] on page G-14</i>	
<i>Vector multiply accumulate long: vmlal_<type>. Vr[i] := Va[i] + Vb[i] * Vc[i] on page G-14</i>	
<i>Vector multiply subtract: vmls{q}_<type>. Vr[i] := Va[i] - Vb[i] * Vc[i] on page G-14</i>	
<i>Vector multiply subtract long on page G-15</i>	
<i>Vector saturating doubling multiply high on page G-15</i>	
<i>Vector saturating rounding doubling multiply high on page G-15</i>	
<i>Vector saturating doubling multiply accumulate long on page G-15</i>	
<i>Vector saturating doubling multiply subtract long on page G-15</i>	
<i>Vector long multiply on page G-15</i>	
<i>Vector saturating doubling long multiply on page G-15</i>	
Subtraction	<i>Vector subtract on page G-16</i>
	<i>Vector long subtract: vsubl_<type>. Vr[i]:=Va[i]-Vb[i] on page G-16</i>
	<i>Vector wide subtract: vsubw_<type>. Vr[i]:=Va[i]-Vb[i] on page G-16</i>
	<i>Vector saturating subtract on page G-16</i>
	<i>Vector halving subtract on page G-17</i>
	<i>Vector subtract high half on page G-17</i>
	<i>Vector rounding subtract high half on page G-17</i>
Comparison	<i>Vector compare equal on page G-18</i>
	<i>Vector compare greater-than or equal on page G-18</i>
	<i>Vector compare less-than or equal on page G-18</i>
	<i>Vector compare greater-than on page G-19</i>

Table G-2 Summary of NEON intrinsics (continued)

Category	Intrinsic description
	<i>Vector compare less-than</i> on page G-19
	<i>Vector compare absolute greater-than or equal</i> on page G-19
	<i>Vector compare absolute less-than or equal</i> on page G-19
	<i>Vector compare absolute greater-than</i> on page G-19
	<i>Vector compare absolute less-than</i> on page G-19
	<i>Vector test bits</i> on page G-19
Absolute difference	<i>Absolute difference between the arguments: $vabd\{q\}_{<type>}$. $Vr[i] = Va[i] - Vb[i]$</i> on page G-21
	<i>Absolute difference - long</i> on page G-21
	<i>Absolute difference and accumulate: $vaba\{q\}_{<type>}$. $Vr[i] = Va[i] + Vb[i] - Vc[i]$</i> on page G-21
	<i>Absolute difference and accumulate - long</i> on page G-21
Max and Min	<i>$vmax\{q\}_{<type>}$. $Vr[i] := (Va[i] >= Vb[i]) ? Va[i] : Vb[i]$</i> on page G-22
	<i>$vmin\{q\}_{<type>}$. $Vr[i] := (Va[i] >= Vb[i]) ? Vb[i] : Va[i]$</i> on page G-22
Pairwise addition	<i>Pairwise add</i> on page G-23
	<i>Long pairwise add</i> on page G-23
	<i>Long pairwise add and accumulate</i> on page G-23
Folding maximum	<i>Folding maximum</i> on page G-24
Folding minimum	<i>Folding minimum</i> on page G-25
Reciprocal and Sqrt	<i>Reciprocal/Sqrt</i> on page G-26
Shifts by signed variable	<i>Vector shift left: $vshl\{q\}_{<type>}$. $Vr[i] := Va[i] << Vb[i]$ (negative values shift right)</i> on page G-27
	<i>Vector saturating shift left: (negative values shift right)</i> on page G-27
	<i>Vector rounding shift left: (negative values shift right)</i> on page G-27
	<i>Vector saturating rounding shift left: (negative values shift right)</i> on page G-28
Shifts by a constant	<i>Vector shift right by constant</i> on page G-29
	<i>Vector shift left by constant</i> on page G-29
	<i>Vector rounding shift right by constant</i> on page G-29
	<i>Vector shift right by constant and accumulate</i> on page G-30
	<i>Vector rounding shift right by constant and accumulate</i> on page G-30
	<i>Vector saturating shift left by constant</i> on page G-30
	<i>Vector signed->unsigned saturating shift left by constant</i> on page G-30
	<i>Vector narrowing shift right by constant</i> on page G-31
	<i>Vector signed->unsigned narrowing saturating shift right by constant</i> on page G-31
	<i>Vector signed->unsigned rounding narrowing saturating shift right by constant</i> on page G-31

Table G-2 Summary of NEON intrinsics (continued)

Category	Intrinsic description
	<i>Vector narrowing saturating shift right by constant on page G-31</i>
	<i>Vector rounding narrowing shift right by constant on page G-31</i>
	<i>Vector rounding narrowing saturating shift right by constant on page G-31</i>
	<i>Vector widening shift left by constant on page G-31</i>
Shifts with insert	<i>Vector shift right and insert on page G-33</i>
	<i>Vector shift left and insert on page G-33</i>
Load vector or lane of a vector	<i>Load a single vector from memory on page G-34</i>
	<i>Initialize a vector from a literal bit pattern on page G-50</i>
	<i>Load a single lane from memory on page G-34</i>
	<i>Load a single lane of a vector from a literal on page G-49</i>
	<i>Load all lanes of vector with same value from memory on page G-35</i>
	<i>Load all lanes of vector to the same literal value on page G-51</i>
	<i>Load all lanes of the vector to the value of a lane of a vector on page G-51</i>
Extract a lane from a vector	<i>Extract lanes from a vector and put into a register on page G-48</i>
Store vector or lane of a vector	<i>Store a single vector into memory on page G-37</i>
	<i>Store a lane of a vector into memory on page G-37</i>
Load N-element structure	<i>Load N-element structure from memory on page G-39</i>
	<i>Load a single lane of N-element structure from memory on page G-41</i>
	<i>Load all lanes of N-element structure with same value from memory on page G-40</i>
Store N-element structure	<i>Store N-element structure to memory on page G-43</i>
	<i>Store a single lane of N-element structure to memory on page G-44</i>
Combining vectors	<i>Combining vectors on page G-53</i>
Splitting vectors	<i>Splitting vectors on page G-54</i>
Converting vectors	<i>Convert from float on page G-55</i>
	<i>Convert to float on page G-55</i>
	<i>Convert between floats on page G-55</i>
	<i>Vector narrow integer on page G-55</i>
	<i>Vector long move on page G-55</i>
	<i>Vector saturating narrow integer on page G-55</i>
	<i>Vector saturating narrow integer signed->unsigned on page G-56</i>

Table G-2 Summary of NEON intrinsics (continued)

Category	Intrinsic description
Table look up	<i>Table look up on page G-57</i>
Extended table look up	<i>Extended table look up intrinsics on page G-58</i>
Operations with a scalar	<i>Vector multiply accumulate with scalar on page G-59</i>
	<i>Vector widening multiply accumulate with scalar on page G-59</i>
	<i>Vector widening saturating doubling multiply accumulate with scalar on page G-59</i>
	<i>Vector multiply subtract with scalar on page G-60</i>
	<i>Vector widening multiply subtract with scalar on page G-60</i>
	<i>Vector widening saturating doubling multiply subtract with scalar on page G-60</i>
	<i>Vector multiply by scalar on page G-60</i>
	<i>Vector long multiply with scalar on page G-61</i>
	<i>Vector long multiply by scalar on page G-61</i>
	<i>Vector saturating doubling long multiply with scalar on page G-61</i>
	<i>Vector saturating doubling long multiply by scalar on page G-61</i>
	<i>Vector saturating doubling multiply high with scalar on page G-61</i>
	<i>Vector saturating doubling multiply high by scalar on page G-61</i>
	<i>Vector saturating rounding doubling multiply high with scalar on page G-62</i>
	<i>Vector rounding saturating doubling multiply high by scalar on page G-62</i>
	<i>Vector multiply accumulate with scalar on page G-62</i>
	<i>Vector widening multiply accumulate with scalar on page G-62</i>
	<i>Vector widening saturating doubling multiply accumulate with scalar on page G-62</i>
	<i>Vector multiply subtract with scalar on page G-62</i>
	<i>Vector widening multiply subtract with scalar on page G-63</i>
<i>Vector widening saturating doubling multiply subtract with scalar on page G-63</i>	
Vector extract	<i>Vector extract on page G-64</i>
Reverse vector elements	<i>Reverse vector elements (swap endianness) on page G-65</i>
Other single operand arithmetic	<i>Absolute: $vabs\{q\}_{<type>}$. $Vd[i] = Va[i]$ on page G-66</i>
	<i>Saturating absolute: $vqabs\{q\}_{<type>}$. $Vd[i] = sat(Va[i])$ on page G-66</i>
	<i>Negate: $vneg\{q\}_{<type>}$. $Vd[i] = -Va[i]$ on page G-66</i>
	<i>Saturating Negate: $vqneg\{q\}_{<type>}$. $sat(Vd[i] = -Va[i])$ on page G-66</i>
	<i>Count leading sign bits on page G-66</i>
	<i>Count leading zeros on page G-66</i>

Table G-2 Summary of NEON intrinsics (continued)

Category	Intrinsic description
	<i>Count number of set bits on page G-67</i>
	<i>Reciprocal estimate on page G-67</i>
	<i>Reciprocal square root estimate on page G-67</i>
Logical operations	<i>Bitwise not on page G-68</i>
	<i>Bitwise and on page G-68</i>
	<i>Bitwise or on page G-68</i>
	<i>Bitwise exclusive or (EOR or XOR) on page G-69</i>
	<i>Bit Clear on page G-69</i>
	<i>Bitwise OR complement on page G-69</i>
	<i>Bitwise Select on page G-70</i>
Transposition operations	<i>Transpose elements on page G-71</i>
	<i>Interleave elements on page G-71</i>
	<i>De-Interleave elements on page G-71</i>

Most NEON instructions have an equivalent NEON intrinsic. The following NEON instructions do not have an equivalent intrinsic:

- VSWP
- VLDM
- VLDR
- VMRS
- VMSR
- VPOP
- VPUSH
- VSTM
- VSTR.

G.4 Intrinsics

The intrinsics described in this topic map closely to NEON instructions. Each topic begins with a list of function prototypes, with a comment specifying an equivalent assembler instruction. The compiler selects an instruction that has the required semantics, but there is no guarantee that the compiler produces the listed instruction.

The intrinsics use a naming scheme that is similar to the NEON unified assembler syntax. That is, each intrinsic has the form:

`<opname><flags>_<type>`

An additional `q` flag is provided to specify that the intrinsic operates on 128-bit vectors.

For example:

- `vmul_s16`, multiplies two vectors of signed 16-bit values.
This compiles to `VMUL.I16 d2, d0, d1`.
- `vaddl_u8`, is a long add of two 64-bit vectors containing unsigned 8-bit values, resulting in a 128-bit vector of unsigned 16-bit values.
This compiles to `VADDL.U8 q1, d0, d1`.

Registers other than those specified in these examples might be used. In addition, the compiler might perform optimization that in some way changes the instruction that the source code compiles to.

Note

The intrinsic function prototypes in this topic use the following type annotations:

`__const(n)` the argument *n* must be a compile-time constant

`__constrange(min, max)`

the argument must be a compile-time constant in the range *min* to *max*

`__transfersize(n)`

the intrinsic loads *n* lanes from this pointer.

Note

The NEON intrinsic function prototypes that use `__fp16` are only available for targets that have the NEON half-precision VFP extension. To enable use of `__fp16`, use the `--fp16_format` command-line option. See [--fp16_format=format](#) on page 3-96.

The intrinsics are grouped into:

- [Addition](#) on page G-12
- [Multiplication](#) on page G-14
- [Subtraction](#) on page G-16
- [Comparison](#) on page G-18
- [Absolute difference](#) on page G-21
- [Max/Min](#) on page G-22
- [Pairwise addition](#) on page G-23
- [Folding maximum](#) on page G-24
- [Folding minimum](#) on page G-25
- [Reciprocal/Sqrt](#) on page G-26
- [Shifts by signed variable](#) on page G-27
- [Shifts by a constant](#) on page G-29

- *Shifts with insert* on page G-33
- *Loads of a single vector or lane* on page G-34
- *Store a single vector or lane* on page G-37
- *Loads of an N-element structure* on page G-39
- *Extract lanes from a vector and put into a register* on page G-48
- *Load a single lane of a vector from a literal* on page G-49
- *Initialize a vector from a literal bit pattern* on page G-50
- *Set all lanes to same value* on page G-51
- *Combining vectors* on page G-53
- *Splitting vectors* on page G-54
- *Converting vectors* on page G-55
- *Table look up* on page G-57
- *Extended table look up intrinsics* on page G-58
- *Operations with a scalar value* on page G-59
- *Vector extract* on page G-64
- *Reverse vector elements (swap endianness)* on page G-65
- *Other single operand arithmetic* on page G-66
- *Logical operations* on page G-68
- *Transposition operations* on page G-71
- *Vector reinterpret cast operations* on page G-73.

G.5 Addition

These intrinsics add vectors. Each lane in the result is the consequence of performing the addition on the corresponding lanes in each operand vector.

G.5.1 Vector add: `vadd{q}_<type>`. `Vr[i]:=Va[i]+Vb[i]`

`Vr`, `Va`, `Vb` have equal lane sizes.

```
int8x8_t   vadd_s8(int8x8_t a, int8x8_t b);           // VADD.I8 d0,d0,d0
int16x4_t  vadd_s16(int16x4_t a, int16x4_t b);      // VADD.I16 d0,d0,d0
int32x2_t  vadd_s32(int32x2_t a, int32x2_t b);      // VADD.I32 d0,d0,d0
int64x1_t  vadd_s64(int64x1_t a, int64x1_t b);      // VADD.I64 d0,d0,d0
float32x2_t vadd_f32(float32x2_t a, float32x2_t b); // VADD.F32 d0,d0,d0
uint8x8_t  vadd_u8(uint8x8_t a, uint8x8_t b);       // VADD.I8 d0,d0,d0
uint16x4_t vadd_u16(uint16x4_t a, uint16x4_t b);    // VADD.I16 d0,d0,d0
uint32x2_t vadd_u32(uint32x2_t a, uint32x2_t b);    // VADD.I32 d0,d0,d0
uint64x1_t vadd_u64(uint64x1_t a, uint64x1_t b);    // VADD.I64 d0,d0,d0
int8x16_t  vaddq_s8(int8x16_t a, int8x16_t b);     // VADD.I8 q0,q0,q0
int16x8_t  vaddq_s16(int16x8_t a, int16x8_t b);     // VADD.I16 q0,q0,q0
int32x4_t  vaddq_s32(int32x4_t a, int32x4_t b);     // VADD.I32 q0,q0,q0
int64x2_t  vaddq_s64(int64x2_t a, int64x2_t b);     // VADD.I64 q0,q0,q0
float32x4_t vaddq_f32(float32x4_t a, float32x4_t b); // VADD.F32 q0,q0,q0
uint8x16_t vaddq_u8(uint8x16_t a, uint8x16_t b);    // VADD.I8 q0,q0,q0
uint16x8_t vaddq_u16(uint16x8_t a, uint16x8_t b);   // VADD.I16 q0,q0,q0
uint32x4_t vaddq_u32(uint32x4_t a, uint32x4_t b);   // VADD.I32 q0,q0,q0
uint64x2_t vaddq_u64(uint64x2_t a, uint64x2_t b);   // VADD.I64 q0,q0,q0
```

G.5.2 Vector long add: `vaddl_<type>`. `Vr[i]:=Va[i]+Vb[i]`

`Va`, `Vb` have equal lane sizes, result is a 128 bit vector of lanes that are twice the width.

```
int16x8_t  vaddl_s8(int8x8_t a, int8x8_t b);        // VADDL.S8 q0,d0,d0
int32x4_t  vaddl_s16(int16x4_t a, int16x4_t b);     // VADDL.S16 q0,d0,d0
int64x2_t  vaddl_s32(int32x2_t a, int32x2_t b);     // VADDL.S32 q0,d0,d0
uint16x8_t vaddl_u8(uint8x8_t a, uint8x8_t b);      // VADDL.U8 q0,d0,d0
uint32x4_t vaddl_u16(uint16x4_t a, uint16x4_t b);   // VADDL.U16 q0,d0,d0
uint64x2_t vaddl_u32(uint32x2_t a, uint32x2_t b);   // VADDL.U32 q0,d0,d0
```

G.5.3 Vector wide add: `vaddw_<type>`. `Vr[i]:=Va[i]+Vb[i]`

```
int16x8_t  vaddw_s8(int16x8_t a, int8x8_t b);       // VADDW.S8 q0,q0,d0
int32x4_t  vaddw_s16(int32x4_t a, int16x4_t b);     // VADDW.S16 q0,q0,d0
int64x2_t  vaddw_s32(int64x2_t a, int32x2_t b);     // VADDW.S32 q0,q0,d0
uint16x8_t vaddw_u8(uint16x8_t a, uint8x8_t b);     // VADDW.U8 q0,q0,d0
uint32x4_t vaddw_u16(uint32x4_t a, uint16x4_t b);   // VADDW.U16 q0,q0,d0
uint64x2_t vaddw_u32(uint64x2_t a, uint32x2_t b);   // VADDW.U32 q0,q0,d0
```

G.5.4 Vector halving add: `vhadd{q}_<type>`. `Vr[i:]=(Va[i]+Vb[i])>>1`

```
int8x8_t   vhadd_s8(int8x8_t a, int8x8_t b);        // VHADD.S8 d0,d0,d0
int16x4_t  vhadd_s16(int16x4_t a, int16x4_t b);     // VHADD.S16 d0,d0,d0
int32x2_t  vhadd_s32(int32x2_t a, int32x2_t b);     // VHADD.S32 d0,d0,d0
uint8x8_t  vhadd_u8(uint8x8_t a, uint8x8_t b);      // VHADD.U8 d0,d0,d0
uint16x4_t vhadd_u16(uint16x4_t a, uint16x4_t b);   // VHADD.U16 d0,d0,d0
uint32x2_t vhadd_u32(uint32x2_t a, uint32x2_t b);   // VHADD.U32 d0,d0,d0
int8x16_t  vhaddq_s8(int8x16_t a, int8x16_t b);    // VHADD.S8 q0,q0,q0
int16x8_t  vhaddq_s16(int16x8_t a, int16x8_t b);   // VHADD.S16 q0,q0,q0
int32x4_t  vhaddq_s32(int32x4_t a, int32x4_t b);   // VHADD.S32 q0,q0,q0
```

```
uint8x16_t vhaddq_u8(uint8x16_t a, uint8x16_t b); // VHADD.U8 q0,q0,q0
uint16x8_t vhaddq_u16(uint16x8_t a, uint16x8_t b); // VHADD.U16 q0,q0,q0
uint32x4_t vhaddq_u32(uint32x4_t a, uint32x4_t b); // VHADD.U32 q0,q0,q0
```

G.5.5 Vector rounding halving add: vrhadd{q}_<type>. Vr[i]:=(Va[i]+Vb[i+1])>>1

```
int8x8_t vrhadd_s8(int8x8_t a, int8x8_t b); // VRHADD.S8 d0,d0,d0
int16x4_t vrhadd_s16(int16x4_t a, int16x4_t b); // VRHADD.S16 d0,d0,d0
int32x2_t vrhadd_s32(int32x2_t a, int32x2_t b); // VRHADD.S32 d0,d0,d0
uint8x8_t vrhadd_u8(uint8x8_t a, uint8x8_t b); // VRHADD.U8 d0,d0,d0
uint16x4_t vrhadd_u16(uint16x4_t a, uint16x4_t b); // VRHADD.U16 d0,d0,d0
uint32x2_t vrhadd_u32(uint32x2_t a, uint32x2_t b); // VRHADD.U32 d0,d0,d0
int8x16_t vrhaddq_s8(int8x16_t a, int8x16_t b); // VRHADD.S8 q0,q0,q0
int16x8_t vrhaddq_s16(int16x8_t a, int16x8_t b); // VRHADD.S16 q0,q0,q0
int32x4_t vrhaddq_s32(int32x4_t a, int32x4_t b); // VRHADD.S32 q0,q0,q0
uint8x16_t vrhaddq_u8(uint8x16_t a, uint8x16_t b); // VRHADD.U8 q0,q0,q0
uint16x8_t vrhaddq_u16(uint16x8_t a, uint16x8_t b); // VRHADD.U16 q0,q0,q0
uint32x4_t vrhaddq_u32(uint32x4_t a, uint32x4_t b); // VRHADD.U32 q0,q0,q0
```

G.5.6 VQADD: Vector saturating add

```
int8x8_t vqadd_s8(int8x8_t a, int8x8_t b); // VQADD.S8 d0,d0,d0
int16x4_t vqadd_s16(int16x4_t a, int16x4_t b); // VQADD.S16 d0,d0,d0
int32x2_t vqadd_s32(int32x2_t a, int32x2_t b); // VQADD.S32 d0,d0,d0
int64x1_t vqadd_s64(int64x1_t a, int64x1_t b); // VQADD.S64 d0,d0,d0
uint8x8_t vqadd_u8(uint8x8_t a, uint8x8_t b); // VQADD.U8 d0,d0,d0
uint16x4_t vqadd_u16(uint16x4_t a, uint16x4_t b); // VQADD.U16 d0,d0,d0
uint32x2_t vqadd_u32(uint32x2_t a, uint32x2_t b); // VQADD.U32 d0,d0,d0
uint64x1_t vqadd_u64(uint64x1_t a, uint64x1_t b); // VQADD.U64 d0,d0,d0
int8x16_t vqaddq_s8(int8x16_t a, int8x16_t b); // VQADD.S8 q0,q0,q0
int16x8_t vqaddq_s16(int16x8_t a, int16x8_t b); // VQADD.S16 q0,q0,q0
int32x4_t vqaddq_s32(int32x4_t a, int32x4_t b); // VQADD.S32 q0,q0,q0
int64x2_t vqaddq_s64(int64x2_t a, int64x2_t b); // VQADD.S64 q0,q0,q0
uint8x16_t vqaddq_u8(uint8x16_t a, uint8x16_t b); // VQADD.U8 q0,q0,q0
uint16x8_t vqaddq_u16(uint16x8_t a, uint16x8_t b); // VQADD.U16 q0,q0,q0
uint32x4_t vqaddq_u32(uint32x4_t a, uint32x4_t b); // VQADD.U32 q0,q0,q0
uint64x2_t vqaddq_u64(uint64x2_t a, uint64x2_t b); // VQADD.U64 q0,q0,q0
```

G.5.7 Vector add high half: vaddhn_<type>. Vr[i]:=Va[i]+Vb[i]

```
int8x8_t vaddhn_s16(int16x8_t a, int16x8_t b); // VADDHN.I16 d0,q0,q0
int16x4_t vaddhn_s32(int32x4_t a, int32x4_t b); // VADDHN.I32 d0,q0,q0
int32x2_t vaddhn_s64(int64x2_t a, int64x2_t b); // VADDHN.I64 d0,q0,q0
uint8x8_t vaddhn_u16(uint16x8_t a, uint16x8_t b); // VADDHN.U16 d0,q0,q0
uint16x4_t vaddhn_u32(uint32x4_t a, uint32x4_t b); // VADDHN.U32 d0,q0,q0
uint32x2_t vaddhn_u64(uint64x2_t a, uint64x2_t b); // VADDHN.U64 d0,q0,q0
```

G.5.8 Vector rounding add high half: vraddhn_<type>.

```
int8x8_t vraddhn_s16(int16x8_t a, int16x8_t b); // VRADDHN.I16 d0,q0,q0
int16x4_t vraddhn_s32(int32x4_t a, int32x4_t b); // VRADDHN.I32 d0,q0,q0
int32x2_t vraddhn_s64(int64x2_t a, int64x2_t b); // VRADDHN.I64 d0,q0,q0
uint8x8_t vraddhn_u16(uint16x8_t a, uint16x8_t b); // VRADDHN.U16 d0,q0,q0
uint16x4_t vraddhn_u32(uint32x4_t a, uint32x4_t b); // VRADDHN.U32 d0,q0,q0
uint32x2_t vraddhn_u64(uint64x2_t a, uint64x2_t b); // VRADDHN.U64 d0,q0,q0
```

G.6 Multiplication

These intrinsics provide operations including multiplication.

G.6.1 Vector multiply: `vmul{q}<type>`. `Vr[i] := Va[i] * Vb[i]`

```
int8x8_t   vmul_s8(int8x8_t a, int8x8_t b);           // VMUL.I8 d0,d0,d0
int16x4_t  vmul_s16(int16x4_t a, int16x4_t b);       // VMUL.I16 d0,d0,d0
int32x2_t  vmul_s32(int32x2_t a, int32x2_t b);       // VMUL.I32 d0,d0,d0
float32x2_t vmul_f32(float32x2_t a, float32x2_t b);  // VMUL.F32 d0,d0,d0
uint8x8_t  vmul_u8(uint8x8_t a, uint8x8_t b);       // VMUL.I8 d0,d0,d0
uint16x4_t vmul_u16(uint16x4_t a, uint16x4_t b);    // VMUL.I16 d0,d0,d0
uint32x2_t vmul_u32(uint32x2_t a, uint32x2_t b);    // VMUL.I32 d0,d0,d0
poly8x8_t  vmul_p8(poly8x8_t a, poly8x8_t b);       // VMUL.P8 d0,d0,d0
int8x16_t  vmulq_s8(int8x16_t a, int8x16_t b);     // VMUL.I8 q0,q0,q0
int16x8_t  vmulq_s16(int16x8_t a, int16x8_t b);    // VMUL.I16 q0,q0,q0
int32x4_t  vmulq_s32(int32x4_t a, int32x4_t b);    // VMUL.I32 q0,q0,q0
float32x4_t vmulq_f32(float32x4_t a, float32x4_t b); // VMUL.F32 q0,q0,q0
int8x16_t  vmulq_u8(int8x16_t a, int8x16_t b);     // VMUL.I8 q0,q0,q0
uint16x8_t vmulq_u16(uint16x8_t a, uint16x8_t b);  // VMUL.I16 q0,q0,q0
uint32x4_t vmulq_u32(uint32x4_t a, uint32x4_t b);  // VMUL.I32 q0,q0,q0
poly8x16_t vmulq_p8(poly8x16_t a, poly8x16_t b);   // VMUL.P8 q0,q0,q0
```

G.6.2 Vector multiply accumulate: `vmla{q}<type>`. `Vr[i] := Va[i] + Vb[i] * Vc[i]`

```
int8x8_t   vmla_s8(int8x8_t a, int8x8_t b, int8x8_t c); // VMLA.I8 d0,d0,d0
int16x4_t  vmla_s16(int16x4_t a, int16x4_t b, int16x4_t c); // VMLA.I16 d0,d0,d0
int32x2_t  vmla_s32(int32x2_t a, int32x2_t b, int32x2_t c); // VMLA.I32 d0,d0,d0
float32x2_t vmla_f32(float32x2_t a, float32x2_t b, float32x2_t c); // VMLA.F32 d0,d0,d0
uint8x8_t  vmla_u8(uint8x8_t a, uint8x8_t b, uint8x8_t c); // VMLA.I8 d0,d0,d0
uint16x4_t vmla_u16(uint16x4_t a, uint16x4_t b, uint16x4_t c); // VMLA.I16 d0,d0,d0
uint32x2_t vmla_u32(uint32x2_t a, uint32x2_t b, uint32x2_t c); // VMLA.I32 d0,d0,d0
int8x16_t  vmlaq_s8(int8x16_t a, int8x16_t b, int8x16_t c); // VMLA.I8 q0,q0,q0
int16x8_t  vmlaq_s16(int16x8_t a, int16x8_t b, int16x8_t c); // VMLA.I16 q0,q0,q0
int32x4_t  vmlaq_s32(int32x4_t a, int32x4_t b, int32x4_t c); // VMLA.I32 q0,q0,q0
float32x4_t vmlaq_f32(float32x4_t a, float32x4_t b, float32x4_t c); // VMLA.F32 q0,q0,q0
uint8x16_t vmlaq_u8(uint8x16_t a, uint8x16_t b, uint8x16_t c); // VMLA.I8 q0,q0,q0
uint16x8_t vmlaq_u16(uint16x8_t a, uint16x8_t b, uint16x8_t c); // VMLA.I16 q0,q0,q0
uint32x4_t vmlaq_u32(uint32x4_t a, uint32x4_t b, uint32x4_t c); // VMLA.I32 q0,q0,q0
```

G.6.3 Vector multiply accumulate long: `vmlal<type>`. `Vr[i] := Va[i] + Vb[i] * Vc[i]`

```
int16x8_t  vmlal_s8(int16x8_t a, int8x8_t b, int8x8_t c); // VMLAL.S8 q0,d0,d0
int32x4_t  vmlal_s16(int32x4_t a, int16x4_t b, int16x4_t c); // VMLAL.S16 q0,d0,d0
int64x2_t  vmlal_s32(int64x2_t a, int32x2_t b, int32x2_t c); // VMLAL.S32 q0,d0,d0
uint16x8_t vmlal_u8(uint16x8_t a, uint8x8_t b, uint8x8_t c); // VMLAL.U8 q0,d0,d0
uint32x4_t vmlal_u16(uint32x4_t a, uint16x4_t b, uint16x4_t c); // VMLAL.U16 q0,d0,d0
uint64x2_t vmlal_u32(uint64x2_t a, uint32x2_t b, uint32x2_t c); // VMLAL.U32 q0,d0,d0
```

G.6.4 Vector multiply subtract: `vmls{q}<type>`. `Vr[i] := Va[i] - Vb[i] * Vc[i]`

```
int8x8_t   vmls_s8(int8x8_t a, int8x8_t b, int8x8_t c); // VMLS.I8 d0,d0,d0
int16x4_t  vmls_s16(int16x4_t a, int16x4_t b, int16x4_t c); // VMLS.I16 d0,d0,d0
int32x2_t  vmls_s32(int32x2_t a, int32x2_t b, int32x2_t c); // VMLS.I32 d0,d0,d0
float32x2_t vmls_f32(float32x2_t a, float32x2_t b, float32x2_t c); // VMLS.F32 d0,d0,d0
uint8x8_t  vmls_u8(uint8x8_t a, uint8x8_t b, uint8x8_t c); // VMLS.I8 d0,d0,d0
uint16x4_t vmls_u16(uint16x4_t a, uint16x4_t b, uint16x4_t c); // VMLS.I16 d0,d0,d0
uint32x2_t vmls_u32(uint32x2_t a, uint32x2_t b, uint32x2_t c); // VMLS.I32 d0,d0,d0
int8x16_t  vmlsq_s8(int8x16_t a, int8x16_t b, int8x16_t c); // VMLS.I8 q0,q0,q0
int16x8_t  vmlsq_s16(int16x8_t a, int16x8_t b, int16x8_t c); // VMLS.I16 q0,q0,q0
int32x4_t  vmlsq_s32(int32x4_t a, int32x4_t b, int32x4_t c); // VMLS.I32 q0,q0,q0
```

```
float32x4_t vmlsq_f32(float32x4_t a, float32x4_t b, float32x4_t c); // VMLS.F32 q0,q0,q0
uint8x16_t vmlsq_u8(uint8x16_t a, uint8x16_t b, uint8x16_t c); // VMLS.I8 q0,q0,q0
uint16x8_t vmlsq_u16(uint16x8_t a, uint16x8_t b, uint16x8_t c); // VMLS.I16 q0,q0,q0
uint32x4_t vmlsq_u32(uint32x4_t a, uint32x4_t b, uint32x4_t c); // VMLS.I32 q0,q0,q0
```

G.6.5 Vector multiply subtract long

```
int16x8_t vmlsl_s8(int16x8_t a, int8x8_t b, int8x8_t c); // VMLSL.S8 q0,d0,d0
int32x4_t vmlsl_s16(int32x4_t a, int16x4_t b, int16x4_t c); // VMLSL.S16 q0,d0,d0
int64x2_t vmlsl_s32(int64x2_t a, int32x2_t b, int32x2_t c); // VMLSL.S32 q0,d0,d0
uint16x8_t vmlsl_u8(uint16x8_t a, uint8x8_t b, uint8x8_t c); // VMLSL.U8 q0,d0,d0
uint32x4_t vmlsl_u16(uint32x4_t a, uint16x4_t b, uint16x4_t c); // VMLSL.U16 q0,d0,d0
uint64x2_t vmlsl_u32(uint64x2_t a, uint32x2_t b, uint32x2_t c); // VMLSL.U32 q0,d0,d0
```

G.6.6 Vector saturating doubling multiply high

```
int16x4_t vqdmulh_s16(int16x4_t a, int16x4_t b); // VQDMULH.S16 d0,d0,d0
int32x2_t vqdmulh_s32(int32x2_t a, int32x2_t b); // VQDMULH.S32 d0,d0,d0
int16x8_t vqdmulhq_s16(int16x8_t a, int16x8_t b); // VQDMULH.S16 q0,q0,q0
int32x4_t vqdmulhq_s32(int32x4_t a, int32x4_t b); // VQDMULH.S32 q0,q0,q0
```

G.6.7 Vector saturating rounding doubling multiply high

```
int16x4_t vqrdmulh_s16(int16x4_t a, int16x4_t b); // VQRDMULH.S16 d0,d0,d0
int32x2_t vqrdmulh_s32(int32x2_t a, int32x2_t b); // VQRDMULH.S32 d0,d0,d0
int16x8_t vqrdmulhq_s16(int16x8_t a, int16x8_t b); // VQRDMULH.S16 q0,q0,q0
int32x4_t vqrdmulhq_s32(int32x4_t a, int32x4_t b); // VQRDMULH.S32 q0,q0,q0
```

G.6.8 Vector saturating doubling multiply accumulate long

```
int32x4_t vqdm1al_s16(int32x4_t a, int16x4_t b, int16x4_t c); // VQDMLAL.S16 q0,d0,d0
int64x2_t vqdm1al_s32(int64x2_t a, int32x2_t b, int32x2_t c); // VQDMLAL.S32 q0,d0,d0
```

G.6.9 Vector saturating doubling multiply subtract long

```
int32x4_t vqdm1sl_s16(int32x4_t a, int16x4_t b, int16x4_t c); // VQDMLSL.S16 q0,d0,d0
int64x2_t vqdm1sl_s32(int64x2_t a, int32x2_t b, int32x2_t c); // VQDMLSL.S32 q0,d0,d0
```

G.6.10 Vector long multiply

```
int16x8_t vmull_s8(int8x8_t a, int8x8_t b); // VMULL.S8 q0,d0,d0
int32x4_t vmull_s16(int16x4_t a, int16x4_t b); // VMULL.S16 q0,d0,d0
int64x2_t vmull_s32(int32x2_t a, int32x2_t b); // VMULL.S32 q0,d0,d0
uint16x8_t vmull_u8(uint8x8_t a, uint8x8_t b); // VMULL.U8 q0,d0,d0
uint32x4_t vmull_u16(uint16x4_t a, uint16x4_t b); // VMULL.U16 q0,d0,d0
uint64x2_t vmull_u32(uint32x2_t a, uint32x2_t b); // VMULL.U32 q0,d0,d0
poly16x8_t vmull_p8(poly8x8_t a, poly8x8_t b); // VMULL.P8 q0,d0,d0
```

G.6.11 Vector saturating doubling long multiply

```
int32x4_t vqdmull_s16(int16x4_t a, int16x4_t b); // VQDMULL.S16 q0,d0,d0
int64x2_t vqdmull_s32(int32x2_t a, int32x2_t b); // VQDMULL.S32 q0,d0,d0
```

G.7 Subtraction

These intrinsics provide operations including subtraction.

G.7.1 Vector subtract

```
int8x8_t   vsub_s8(int8x8_t a, int8x8_t b);           // VSUB.I8 d0,d0,d0
int16x4_t  vsub_s16(int16x4_t a, int16x4_t b);      // VSUB.I16 d0,d0,d0
int32x2_t  vsub_s32(int32x2_t a, int32x2_t b);      // VSUB.I32 d0,d0,d0
int64x1_t  vsub_s64(int64x1_t a, int64x1_t b);      // VSUB.I64 d0,d0,d0
float32x2_t vsub_f32(float32x2_t a, float32x2_t b); // VSUB.F32 d0,d0,d0
uint8x8_t  vsub_u8(uint8x8_t a, uint8x8_t b);       // VSUB.I8 d0,d0,d0
uint16x4_t vsub_u16(uint16x4_t a, uint16x4_t b);    // VSUB.I16 d0,d0,d0
uint32x2_t vsub_u32(uint32x2_t a, uint32x2_t b);    // VSUB.I32 d0,d0,d0
uint64x1_t vsub_u64(uint64x1_t a, uint64x1_t b);    // VSUB.I64 d0,d0,d0
int8x16_t  vsubq_s8(int8x16_t a, int8x16_t b);      // VSUB.I8 q0,q0,q0
int16x8_t  vsubq_s16(int16x8_t a, int16x8_t b);     // VSUB.I16 q0,q0,q0
int32x4_t  vsubq_s32(int32x4_t a, int32x4_t b);     // VSUB.I32 q0,q0,q0
int64x2_t  vsubq_s64(int64x2_t a, int64x2_t b);     // VSUB.I64 q0,q0,q0
float32x4_t vsubq_f32(float32x4_t a, float32x4_t b); // VSUB.F32 q0,q0,q0
uint8x16_t vsubq_u8(uint8x16_t a, uint8x16_t b);    // VSUB.I8 q0,q0,q0
uint16x8_t vsubq_u16(uint16x8_t a, uint16x8_t b);    // VSUB.I16 q0,q0,q0
uint32x4_t vsubq_u32(uint32x4_t a, uint32x4_t b);    // VSUB.I32 q0,q0,q0
uint64x2_t vsubq_u64(uint64x2_t a, uint64x2_t b);    // VSUB.I64 q0,q0,q0
```

G.7.2 Vector long subtract: `vsubl_<type>`. `Vr[i]:=Va[i]+Vb[i]`

```
int16x8_t  vsubl_s8(int8x8_t a, int8x8_t b);        // VSUBL.S8 q0,d0,d0
int32x4_t  vsubl_s16(int16x4_t a, int16x4_t b);     // VSUBL.S16 q0,d0,d0
int64x2_t  vsubl_s32(int32x2_t a, int32x2_t b);     // VSUBL.S32 q0,d0,d0
uint16x8_t vsubl_u8(uint8x8_t a, uint8x8_t b);      // VSUBL.U8 q0,d0,d0
uint32x4_t vsubl_u16(uint16x4_t a, uint16x4_t b);   // VSUBL.U16 q0,d0,d0
uint64x2_t vsubl_u32(uint32x2_t a, uint32x2_t b);   // VSUBL.U32 q0,d0,d0
```

G.7.3 Vector wide subtract: `vsubw_<type>`. `Vr[i]:=Va[i]+Vb[i]`

```
int16x8_t  vsubw_s8(int16x8_t a, int8x8_t b);       // VSUBW.S8 q0,q0,d0
int32x4_t  vsubw_s16(int32x4_t a, int16x4_t b);     // VSUBW.S16 q0,q0,d0
int64x2_t  vsubw_s32(int64x2_t a, int32x2_t b);     // VSUBW.S32 q0,q0,d0
uint16x8_t vsubw_u8(uint16x8_t a, uint8x8_t b);     // VSUBW.U8 q0,q0,d0
uint32x4_t vsubw_u16(uint32x4_t a, uint16x4_t b);   // VSUBW.U16 q0,q0,d0
uint64x2_t vsubw_u32(uint64x2_t a, uint32x2_t b);   // VSUBW.U32 q0,q0,d0
```

G.7.4 Vector saturating subtract

```
int8x8_t   vqsub_s8(int8x8_t a, int8x8_t b);        // VQSUB.S8 d0,d0,d0
int16x4_t  vqsub_s16(int16x4_t a, int16x4_t b);    // VQSUB.S16 d0,d0,d0
int32x2_t  vqsub_s32(int32x2_t a, int32x2_t b);    // VQSUB.S32 d0,d0,d0
int64x1_t  vqsub_s64(int64x1_t a, int64x1_t b);    // VQSUB.S64 d0,d0,d0
uint8x8_t  vqsub_u8(uint8x8_t a, uint8x8_t b);     // VQSUB.U8 d0,d0,d0
uint16x4_t vqsub_u16(uint16x4_t a, uint16x4_t b);  // VQSUB.U16 d0,d0,d0
uint32x2_t vqsub_u32(uint32x2_t a, uint32x2_t b);  // VQSUB.U32 d0,d0,d0
uint64x1_t vqsub_u64(uint64x1_t a, uint64x1_t b);  // VQSUB.U64 d0,d0,d0
int8x16_t  vqsubq_s8(int8x16_t a, int8x16_t b);    // VQSUB.S8 q0,q0,q0
int16x8_t  vqsubq_s16(int16x8_t a, int16x8_t b);   // VQSUB.S16 q0,q0,q0
int32x4_t  vqsubq_s32(int32x4_t a, int32x4_t b);   // VQSUB.S32 q0,q0,q0
int64x2_t  vqsubq_s64(int64x2_t a, int64x2_t b);   // VQSUB.S64 q0,q0,q0
uint8x16_t vqsubq_u8(uint8x16_t a, uint8x16_t b);  // VQSUB.U8 q0,q0,q0
uint16x8_t vqsubq_u16(uint16x8_t a, uint16x8_t b);  // VQSUB.U16 q0,q0,q0
uint32x4_t vqsubq_u32(uint32x4_t a, uint32x4_t b);  // VQSUB.U32 q0,q0,q0
uint64x2_t vqsubq_u64(uint64x2_t a, uint64x2_t b);  // VQSUB.U64 q0,q0,q0
```

G.7.5 Vector halving subtract

```

int8x8_t  vhsb_s8(int8x8_t a, int8x8_t b);    // VHSUB.S8 d0,d0,d0
int16x4_t vhsb_s16(int16x4_t a, int16x4_t b); // VHSUB.S16 d0,d0,d0
int32x2_t vhsb_s32(int32x2_t a, int32x2_t b); // VHSUB.S32 d0,d0,d0
uint8x8_t vhsb_u8(uint8x8_t a, uint8x8_t b);  // VHSUB.U8 d0,d0,d0
uint16x4_t vhsb_u16(uint16x4_t a, uint16x4_t b); // VHSUB.U16 d0,d0,d0
uint32x2_t vhsb_u32(uint32x2_t a, uint32x2_t b); // VHSUB.U32 d0,d0,d0
int8x16_t vhsbq_s8(int8x16_t a, int8x16_t b); // VHSUB.S8 q0,q0,q0
int16x8_t vhsbq_s16(int16x8_t a, int16x8_t b); // VHSUB.S16 q0,q0,q0
int32x4_t vhsbq_s32(int32x4_t a, int32x4_t b); // VHSUB.S32 q0,q0,q0
uint8x16_t vhsbq_u8(uint8x16_t a, uint8x16_t b); // VHSUB.U8 q0,q0,q0
uint16x8_t vhsbq_u16(uint16x8_t a, uint16x8_t b); // VHSUB.U16 q0,q0,q0
uint32x4_t vhsbq_u32(uint32x4_t a, uint32x4_t b); // VHSUB.U32 q0,q0,q0

```

G.7.6 Vector subtract high half

```

int8x8_t  vsubhn_s16(int16x8_t a, int16x8_t b); // VSUBHN.I16 d0,q0,q0
int16x4_t vsubhn_s32(int32x4_t a, int32x4_t b); // VSUBHN.I32 d0,q0,q0
int32x2_t vsubhn_s64(int64x2_t a, int64x2_t b); // VSUBHN.I64 d0,q0,q0
uint8x8_t vsubhn_u16(uint16x8_t a, uint16x8_t b); // VSUBHN.I16 d0,q0,q0
uint16x4_t vsubhn_u32(uint32x4_t a, uint32x4_t b); // VSUBHN.I32 d0,q0,q0
uint32x2_t vsubhn_u64(uint64x2_t a, uint64x2_t b); // VSUBHN.I64 d0,q0,q0

```

G.7.7 Vector rounding subtract high half

```

int8x8_t  vrsubhn_s16(int16x8_t a, int16x8_t b); // VRSUBHN.I16 d0,q0,q0
int16x4_t vrsubhn_s32(int32x4_t a, int32x4_t b); // VRSUBHN.I32 d0,q0,q0
int32x2_t vrsubhn_s64(int64x2_t a, int64x2_t b); // VRSUBHN.I64 d0,q0,q0
uint8x8_t vrsubhn_u16(uint16x8_t a, uint16x8_t b); // VRSUBHN.I16 d0,q0,q0
uint16x4_t vrsubhn_u32(uint32x4_t a, uint32x4_t b); // VRSUBHN.I32 d0,q0,q0
uint32x2_t vrsubhn_u64(uint64x2_t a, uint64x2_t b); // VRSUBHN.I64 d0,q0,q0

```

G.8 Comparison

A range of comparison intrinsics are provided. If the comparison is true for a lane, the result in that lane is all bits set to one. If the comparison is false for a lane, all bits are set to zero. The return type is an unsigned integer type. This means that you can use the result of a comparison as the first argument for the vbsl intrinsics.

G.8.1 Vector compare equal

```
uint8x8_t vceq_s8(int8x8_t a, int8x8_t b);           // VCEQ.I8 d0, d0, d0
uint16x4_t vceq_s16(int16x4_t a, int16x4_t b);      // VCEQ.I16 d0, d0, d0
uint32x2_t vceq_s32(int32x2_t a, int32x2_t b);      // VCEQ.I32 d0, d0, d0
uint32x2_t vceq_f32(float32x2_t a, float32x2_t b);  // VCEQ.F32 d0, d0, d0
uint8x8_t vceq_u8(uint8x8_t a, uint8x8_t b);       // VCEQ.I8 d0, d0, d0
uint16x4_t vceq_u16(uint16x4_t a, uint16x4_t b);   // VCEQ.I16 d0, d0, d0
uint32x2_t vceq_u32(uint32x2_t a, uint32x2_t b);   // VCEQ.I32 d0, d0, d0
uint8x8_t vceq_p8(poly8x8_t a, poly8x8_t b);       // VCEQ.I8 d0, d0, d0
uint8x16_t vceq_s8(int8x16_t a, int8x16_t b);      // VCEQ.I8 q0, q0, q0
uint16x8_t vceq_s16(int16x8_t a, int16x8_t b);     // VCEQ.I16 q0, q0, q0
uint32x4_t vceq_s32(int32x4_t a, int32x4_t b);     // VCEQ.I32 q0, q0, q0
uint32x4_t vceq_f32(float32x4_t a, float32x4_t b); // VCEQ.F32 q0, q0, q0
uint8x16_t vceq_u8(uint8x16_t a, uint8x16_t b);    // VCEQ.I8 q0, q0, q0
uint16x8_t vceq_u16(uint16x8_t a, uint16x8_t b);   // VCEQ.I16 q0, q0, q0
uint32x4_t vceq_u32(uint32x4_t a, uint32x4_t b);   // VCEQ.I32 q0, q0, q0
uint8x16_t vceq_p8(poly8x16_t a, poly8x16_t b);    // VCEQ.I8 q0, q0, q0
```

G.8.2 Vector compare greater-than or equal

```
uint8x8_t vcge_s8(int8x8_t a, int8x8_t b);         // VCGE.S8 d0, d0, d0
uint16x4_t vcge_s16(int16x4_t a, int16x4_t b);     // VCGE.S16 d0, d0, d0
uint32x2_t vcge_s32(int32x2_t a, int32x2_t b);     // VCGE.S32 d0, d0, d0
uint32x2_t vcge_f32(float32x2_t a, float32x2_t b); // VCGE.F32 d0, d0, d0
uint8x8_t vcge_u8(uint8x8_t a, uint8x8_t b);       // VCGE.U8 d0, d0, d0
uint16x4_t vcge_u16(uint16x4_t a, uint16x4_t b);   // VCGE.U16 d0, d0, d0
uint32x2_t vcge_u32(uint32x2_t a, uint32x2_t b);   // VCGE.U32 d0, d0, d0
uint8x16_t vcgeq_s8(int8x16_t a, int8x16_t b);    // VCGE.S8 q0, q0, q0
uint16x8_t vcgeq_s16(int16x8_t a, int16x8_t b);   // VCGE.S16 q0, q0, q0
uint32x4_t vcgeq_s32(int32x4_t a, int32x4_t b);   // VCGE.S32 q0, q0, q0
uint32x4_t vcgeq_f32(float32x4_t a, float32x4_t b); // VCGE.F32 q0, q0, q0
uint8x16_t vcgeq_u8(uint8x16_t a, uint8x16_t b);  // VCGE.U8 q0, q0, q0
uint16x8_t vcgeq_u16(uint16x8_t a, uint16x8_t b); // VCGE.U16 q0, q0, q0
uint32x4_t vcgeq_u32(uint32x4_t a, uint32x4_t b); // VCGE.U32 q0, q0, q0
```

G.8.3 Vector compare less-than or equal

```
uint8x8_t vcle_s8(int8x8_t a, int8x8_t b);         // VCGE.S8 d0, d0, d0
uint16x4_t vcle_s16(int16x4_t a, int16x4_t b);     // VCGE.S16 d0, d0, d0
uint32x2_t vcle_s32(int32x2_t a, int32x2_t b);     // VCGE.S32 d0, d0, d0
uint32x2_t vcle_f32(float32x2_t a, float32x2_t b); // VCGE.F32 d0, d0, d0
uint8x8_t vcle_u8(uint8x8_t a, uint8x8_t b);       // VCGE.U8 d0, d0, d0
uint16x4_t vcle_u16(uint16x4_t a, uint16x4_t b);   // VCGE.U16 d0, d0, d0
uint32x2_t vcle_u32(uint32x2_t a, uint32x2_t b);   // VCGE.U32 d0, d0, d0
uint8x16_t vcleq_s8(int8x16_t a, int8x16_t b);    // VCGE.S8 q0, q0, q0
uint16x8_t vcleq_s16(int16x8_t a, int16x8_t b);   // VCGE.S16 q0, q0, q0
uint32x4_t vcleq_s32(int32x4_t a, int32x4_t b);   // VCGE.S32 q0, q0, q0
uint32x4_t vcleq_f32(float32x4_t a, float32x4_t b); // VCGE.F32 q0, q0, q0
uint8x16_t vcleq_u8(uint8x16_t a, uint8x16_t b);  // VCGE.U8 q0, q0, q0
uint16x8_t vcleq_u16(uint16x8_t a, uint16x8_t b); // VCGE.U16 q0, q0, q0
uint32x4_t vcleq_u32(uint32x4_t a, uint32x4_t b); // VCGE.U32 q0, q0, q0
```


G.8.4 Vector compare greater-than

```

uint8x8_t vcgt_s8(int8x8_t a, int8x8_t b); // VCGT.S8 d0, d0, d0
uint16x4_t vcgt_s16(int16x4_t a, int16x4_t b); // VCGT.S16 d0, d0, d0
uint32x2_t vcgt_s32(int32x2_t a, int32x2_t b); // VCGT.S32 d0, d0, d0
uint32x2_t vcgt_f32(float32x2_t a, float32x2_t b); // VCGT.F32 d0, d0, d0
uint8x8_t vcgt_u8(uint8x8_t a, uint8x8_t b); // VCGT.U8 d0, d0, d0
uint16x4_t vcgt_u16(uint16x4_t a, uint16x4_t b); // VCGT.U16 d0, d0, d0
uint32x2_t vcgt_u32(uint32x2_t a, uint32x2_t b); // VCGT.U32 d0, d0, d0
uint8x16_t vcgtq_s8(int8x16_t a, int8x16_t b); // VCGT.S8 q0, q0, q0
uint16x8_t vcgtq_s16(int16x8_t a, int16x8_t b); // VCGT.S16 q0, q0, q0
uint32x4_t vcgtq_s32(int32x4_t a, int32x4_t b); // VCGT.S32 q0, q0, q0
uint8x16_t vcgtq_u8(uint8x16_t a, uint8x16_t b); // VCGT.U8 q0, q0, q0
uint16x8_t vcgtq_u16(uint16x8_t a, uint16x8_t b); // VCGT.U16 q0, q0, q0
uint32x4_t vcgtq_u32(uint32x4_t a, uint32x4_t b); // VCGT.U32 q0, q0, q0

```

G.8.5 Vector compare less-than

```

uint8x8_t vclt_s8(int8x8_t a, int8x8_t b); // VCGT.S8 d0, d0, d0
uint16x4_t vclt_s16(int16x4_t a, int16x4_t b); // VCGT.S16 d0, d0, d0
uint32x2_t vclt_s32(int32x2_t a, int32x2_t b); // VCGT.S32 d0, d0, d0
uint32x2_t vclt_f32(float32x2_t a, float32x2_t b); // VCGT.F32 d0, d0, d0
uint8x8_t vclt_u8(uint8x8_t a, uint8x8_t b); // VCGT.U8 d0, d0, d0
uint16x4_t vclt_u16(uint16x4_t a, uint16x4_t b); // VCGT.U16 d0, d0, d0
uint32x2_t vclt_u32(uint32x2_t a, uint32x2_t b); // VCGT.U32 d0, d0, d0
uint8x16_t vcltq_s8(int8x16_t a, int8x16_t b); // VCGT.S8 q0, q0, q0
uint16x8_t vcltq_s16(int16x8_t a, int16x8_t b); // VCGT.S16 q0, q0, q0
uint32x4_t vcltq_s32(int32x4_t a, int32x4_t b); // VCGT.S32 q0, q0, q0
uint32x4_t vcltq_f32(float32x4_t a, float32x4_t b); // VCGT.F32 q0, q0, q0
uint8x16_t vcltq_u8(uint8x16_t a, uint8x16_t b); // VCGT.U8 q0, q0, q0
uint16x8_t vcltq_u16(uint16x8_t a, uint16x8_t b); // VCGT.U16 q0, q0, q0
uint32x4_t vcltq_u32(uint32x4_t a, uint32x4_t b); // VCGT.U32 q0, q0, q0

```

G.8.6 Vector compare absolute greater-than or equal

```

uint32x2_t vacge_f32(float32x2_t a, float32x2_t b); // VACGE.F32 d0, d0, d0
uint32x4_t vacgeq_f32(float32x4_t a, float32x4_t b); // VACGE.F32 q0, q0, q0

```

G.8.7 Vector compare absolute less-than or equal

```

uint32x2_t vacle_f32(float32x2_t a, float32x2_t b); // VACGE.F32 d0, d0, d0
uint32x4_t vacleq_f32(float32x4_t a, float32x4_t b); // VACGE.F32 q0, q0, q0

```

G.8.8 Vector compare absolute greater-than

```

uint32x2_t vacgt_f32(float32x2_t a, float32x2_t b); // VACGT.F32 d0, d0, d0
uint32x4_t vacgtq_f32(float32x4_t a, float32x4_t b); // VACGT.F32 q0, q0, q0

```

G.8.9 Vector compare absolute less-than

```

uint32x2_t vaclt_f32(float32x2_t a, float32x2_t b); // VACGT.F32 d0, d0, d0
uint32x4_t vacltq_f32(float32x4_t a, float32x4_t b); // VACGT.F32 q0, q0, q0

```

G.8.10 Vector test bits

```

uint8x8_t vtst_s8(int8x8_t a, int8x8_t b); // VTST.8 d0, d0, d0
uint16x4_t vtst_s16(int16x4_t a, int16x4_t b); // VTST.16 d0, d0, d0
uint32x2_t vtst_s32(int32x2_t a, int32x2_t b); // VTST.32 d0, d0, d0
uint8x8_t vtst_u8(uint8x8_t a, uint8x8_t b); // VTST.8 d0, d0, d0
uint16x4_t vtst_u16(uint16x4_t a, uint16x4_t b); // VTST.16 d0, d0, d0

```

```
uint32x2_t vtst_u32(uint32x2_t a, uint32x2_t b); // VTST.32 d0, d0, d0
uint8x8_t vtst_p8(poly8x8_t a, poly8x8_t b); // VTST.8 d0, d0, d0
uint8x16_t vtstq_s8(int8x16_t a, int8x16_t b); // VTST.8 q0, q0, q0
uint16x8_t vtstq_s16(int16x8_t a, int16x8_t b); // VTST.16 q0, q0, q0
uint32x4_t vtstq_s32(int32x4_t a, int32x4_t b); // VTST.32 q0, q0, q0
uint8x16_t vtstq_u8(uint8x16_t a, uint8x16_t b); // VTST.8 q0, q0, q0
uint16x8_t vtstq_u16(uint16x8_t a, uint16x8_t b); // VTST.16 q0, q0, q0
uint32x4_t vtstq_u32(uint32x4_t a, uint32x4_t b); // VTST.32 q0, q0, q0
uint8x16_t vtstq_p8(poly8x16_t a, poly8x16_t b); // VTST.8 q0, q0, q0
```

G.9 Absolute difference

These intrinsics provide operations including absolute difference.

G.9.1 Absolute difference between the arguments: `vabd{q}_<type>`. $Vr[i] = |Va[i] - Vb[i]|$

```
int8x8_t   vabd_s8(int8x8_t a, int8x8_t b);           // VABD.S8 d0,d0,d0
int16x4_t  vabd_s16(int16x4_t a, int16x4_t b);       // VABD.S16 d0,d0,d0
int32x2_t  vabd_s32(int32x2_t a, int32x2_t b);       // VABD.S32 d0,d0,d0
uint8x8_t  vabd_u8(uint8x8_t a, uint8x8_t b);        // VABD.U8 d0,d0,d0
uint16x4_t vabd_u16(uint16x4_t a, uint16x4_t b);     // VABD.U16 d0,d0,d0
uint32x2_t vabd_u32(uint32x2_t a, uint32x2_t b);     // VABD.U32 d0,d0,d0
float32x2_t vabd_f32(float32x2_t a, float32x2_t b);  // VABD.F32 d0,d0,d0
int8x16_t  vabdq_s8(int8x16_t a, int8x16_t b);      // VABD.S8 q0,q0,q0
int16x8_t  vabdq_s16(int16x8_t a, int16x8_t b);     // VABD.S16 q0,q0,q0
int32x4_t  vabdq_s32(int32x4_t a, int32x4_t b);     // VABD.S32 q0,q0,q0
uint8x16_t vabdq_u8(uint8x16_t a, uint8x16_t b);    // VABD.U8 q0,q0,q0
uint16x8_t vabdq_u16(uint16x8_t a, uint16x8_t b);   // VABD.U16 q0,q0,q0
uint32x4_t vabdq_u32(uint32x4_t a, uint32x4_t b);   // VABD.U32 q0,q0,q0
float32x4_t vabdq_f32(float32x4_t a, float32x4_t b); // VABD.F32 q0,q0,q0
```

G.9.2 Absolute difference - long

```
int16x8_t  vabdl_s8(int8x8_t a, int8x8_t b);        // VABDL.S8 q0,d0,d0
int32x4_t  vabdl_s16(int16x4_t a, int16x4_t b);     // VABDL.S16 q0,d0,d0
int64x2_t  vabdl_s32(int32x2_t a, int32x2_t b);     // VABDL.S32 q0,d0,d0
uint16x8_t vabdl_u8(uint8x8_t a, uint8x8_t b);      // VABDL.U8 q0,d0,d0
uint32x4_t vabdl_u16(uint16x4_t a, uint16x4_t b);   // VABDL.U16 q0,d0,d0
uint64x2_t vabdl_u32(uint32x2_t a, uint32x2_t b);   // VABDL.U32 q0,d0,d0
```

G.9.3 Absolute difference and accumulate: `vaba{q}_<type>`. $Vr[i] = Va[i] + |Vb[i] - Vc[i]|$

```
int8x8_t   vaba_s8(int8x8_t a, int8x8_t b, int8x8_t c); // VABA.S8 d0,d0,d0
int16x4_t  vaba_s16(int16x4_t a, int16x4_t b, int16x4_t c); // VABA.S16 d0,d0,d0
int32x2_t  vaba_s32(int32x2_t a, int32x2_t b, int32x2_t c); // VABA.S32 d0,d0,d0
uint8x8_t  vaba_u8(uint8x8_t a, uint8x8_t b, uint8x8_t c); // VABA.U8 d0,d0,d0
uint16x4_t vaba_u16(uint16x4_t a, uint16x4_t b, uint16x4_t c); // VABA.U16 d0,d0,d0
uint32x2_t vaba_u32(uint32x2_t a, uint32x2_t b, uint32x2_t c); // VABA.U32 d0,d0,d0
int8x16_t  vabaq_s8(int8x16_t a, int8x16_t b, int8x16_t c); // VABA.S8 q0,q0,q0
int16x8_t  vabaq_s16(int16x8_t a, int16x8_t b, int16x8_t c); // VABA.S16 q0,q0,q0
int32x4_t  vabaq_s32(int32x4_t a, int32x4_t b, int32x4_t c); // VABA.S32 q0,q0,q0
uint8x16_t vabaq_u8(uint8x16_t a, uint8x16_t b, uint8x16_t c); // VABA.U8 q0,q0,q0
uint16x8_t vabaq_u16(uint16x8_t a, uint16x8_t b, uint16x8_t c); // VABA.U16 q0,q0,q0
uint32x4_t vabaq_u32(uint32x4_t a, uint32x4_t b, uint32x4_t c); // VABA.U32 q0,q0,q0
```

G.9.4 Absolute difference and accumulate - long

```
int16x8_t  vabal_s8(int16x8_t a, int8x8_t b, int8x8_t c); // VABAL.S8 q0,d0,d0
int32x4_t  vabal_s16(int32x4_t a, int16x4_t b, int16x4_t c); // VABAL.S16 q0,d0,d0
int64x2_t  vabal_s32(int64x2_t a, int32x2_t b, int32x2_t c); // VABAL.S32 q0,d0,d0
uint16x8_t vabal_u8(uint16x8_t a, uint8x8_t b, uint8x8_t c); // VABAL.U8 q0,d0,d0
uint32x4_t vabal_u16(uint32x4_t a, uint16x4_t b, uint16x4_t c); // VABAL.U16 q0,d0,d0
uint64x2_t vabal_u32(uint64x2_t a, uint32x2_t b, uint32x2_t c); // VABAL.U32 q0,d0,d0
```

G.10 Max/Min

These intrinsics provide maximum and minimum operations.

G.10.1 `vmax{q}_<type>. Vr[i] := (Va[i] >= Vb[i]) ? Va[i] : Vb[i]`

```
int8x8_t    vmax_s8(int8x8_t a, int8x8_t b);        // VMAX.S8 d0,d0,d0
int16x4_t   vmax_s16(int16x4_t a, int16x4_t b);    // VMAX.S16 d0,d0,d0
int32x2_t   vmax_s32(int32x2_t a, int32x2_t b);    // VMAX.S32 d0,d0,d0
uint8x8_t   vmax_u8(uint8x8_t a, uint8x8_t b);     // VMAX.U8 d0,d0,d0
uint16x4_t  vmax_u16(uint16x4_t a, uint16x4_t b);  // VMAX.U16 d0,d0,d0
uint32x2_t  vmax_u32(uint32x2_t a, uint32x2_t b);  // VMAX.U32 d0,d0,d0
float32x2_t vmax_f32(float32x2_t a, float32x2_t b); // VMAX.F32 d0,d0,d0
int8x16_t   vmaxq_s8(int8x16_t a, int8x16_t b);   // VMAX.S8 q0,q0,q0
int16x8_t   vmaxq_s16(int16x8_t a, int16x8_t b);   // VMAX.S16 q0,q0,q0
int32x4_t   vmaxq_s32(int32x4_t a, int32x4_t b);   // VMAX.S32 q0,q0,q0
uint8x16_t  vmaxq_u8(uint8x16_t a, uint8x16_t b);  // VMAX.U8 q0,q0,q0
uint16x8_t  vmaxq_u16(uint16x8_t a, uint16x8_t b); // VMAX.U16 q0,q0,q0
uint32x4_t  vmaxq_u32(uint32x4_t a, uint32x4_t b); // VMAX.U32 q0,q0,q0
float32x4_t vmaxq_f32(float32x4_t a, float32x4_t b); // VMAX.F32 q0,q0,q0
```

G.10.2 `vmin{q}_<type>. Vr[i] := (Va[i] >= Vb[i]) ? Vb[i] : Va[i]`

```
int8x8_t    vmin_s8(int8x8_t a, int8x8_t b);      // VMIN.S8 d0,d0,d0
int16x4_t   vmin_s16(int16x4_t a, int16x4_t b);   // VMIN.S16 d0,d0,d0
int32x2_t   vmin_s32(int32x2_t a, int32x2_t b);   // VMIN.S32 d0,d0,d0
uint8x8_t   vmin_u8(uint8x8_t a, uint8x8_t b);    // VMIN.U8 d0,d0,d0
uint16x4_t  vmin_u16(uint16x4_t a, uint16x4_t b); // VMIN.U16 d0,d0,d0
uint32x2_t  vmin_u32(uint32x2_t a, uint32x2_t b); // VMIN.U32 d0,d0,d0
float32x2_t vmin_f32(float32x2_t a, float32x2_t b); // VMIN.F32 d0,d0,d0
int8x16_t   vminq_s8(int8x16_t a, int8x16_t b);   // VMIN.S8 q0,q0,q0
int16x8_t   vminq_s16(int16x8_t a, int16x8_t b);   // VMIN.S16 q0,q0,q0
int32x4_t   vminq_s32(int32x4_t a, int32x4_t b);   // VMIN.S32 q0,q0,q0
uint8x16_t  vminq_u8(uint8x16_t a, uint8x16_t b);  // VMIN.U8 q0,q0,q0
uint16x8_t  vminq_u16(uint16x8_t a, uint16x8_t b); // VMIN.U16 q0,q0,q0
uint32x4_t  vminq_u32(uint32x4_t a, uint32x4_t b); // VMIN.U32 q0,q0,q0
float32x4_t vminq_f32(float32x4_t a, float32x4_t b); // VMIN.F32 q0,q0,q0
```

G.11 Pairwise addition

These intrinsics provide pairwise addition operations.

G.11.1 Pairwise add

```
int8x8_t    vpadd_s8(int8x8_t a, int8x8_t b);    // VPADD.I8 d0,d0,d0
int16x4_t  vpadd_s16(int16x4_t a, int16x4_t b); // VPADD.I16 d0,d0,d0
int32x2_t  vpadd_s32(int32x2_t a, int32x2_t b); // VPADD.I32 d0,d0,d0
uint8x8_t  vpadd_u8(uint8x8_t a, uint8x8_t b);  // VPADD.I8 d0,d0,d0
uint16x4_t vpadd_u16(uint16x4_t a, uint16x4_t b); // VPADD.I16 d0,d0,d0
uint32x2_t vpadd_u32(uint32x2_t a, uint32x2_t b); // VPADD.I32 d0,d0,d0
float32x2_t vpadd_f32(float32x2_t a, float32x2_t b); // VPADD.F32 d0,d0,d0
```

G.11.2 Long pairwise add

```
int16x4_t  vpaddl_s8(int8x8_t a);    // VPADDL.S8 d0,d0
int32x2_t  vpaddl_s16(int16x4_t a);  // VPADDL.S16 d0,d0
int64x1_t  vpaddl_s32(int32x2_t a);  // VPADDL.S32 d0,d0
uint16x4_t vpaddl_u8(uint8x8_t a);   // VPADDL.U8 d0,d0
uint32x2_t vpaddl_u16(uint16x4_t a); // VPADDL.U16 d0,d0
uint64x1_t vpaddl_u32(uint32x2_t a); // VPADDL.U32 d0,d0
int16x8_t  vpaddlq_s8(int8x16_t a);  // VPADDL.S8 q0,q0
int32x4_t  vpaddlq_s16(int16x8_t a); // VPADDL.S16 q0,q0
int64x2_t  vpaddlq_s32(int32x4_t a); // VPADDL.S32 q0,q0
uint16x8_t vpaddlq_u8(uint8x16_t a); // VPADDL.U8 q0,q0
uint32x4_t vpaddlq_u16(uint16x8_t a); // VPADDL.U16 q0,q0
uint64x2_t vpaddlq_u32(uint32x4_t a); // VPADDL.U32 q0,q0
```

G.11.3 Long pairwise add and accumulate

```
int16x4_t  vpadal_s8(int16x4_t a, int8x8_t b);    // VPADAL.S8 d0,d0
int32x2_t  vpadal_s16(int32x2_t a, int16x4_t b); // VPADAL.S16 d0,d0
int64x1_t  vpadal_s32(int64x1_t a, int32x2_t b); // VPADAL.S32 d0,d0
uint16x4_t vpadal_u8(uint16x4_t a, uint8x8_t b);  // VPADAL.U8 d0,d0
uint32x2_t vpadal_u16(uint32x2_t a, uint16x4_t b); // VPADAL.U16 d0,d0
uint64x1_t vpadal_u32(uint64x1_t a, uint32x2_t b); // VPADAL.U32 d0,d0
int16x8_t  vpadalq_s8(int16x8_t a, int8x16_t b); // VPADAL.S8 q0,q0
int32x4_t  vpadalq_s16(int32x4_t a, int16x8_t b); // VPADAL.S16 q0,q0
int64x2_t  vpadalq_s32(int64x2_t a, int32x4_t b); // VPADAL.S32 q0,q0
uint16x8_t vpadalq_u8(uint16x8_t a, uint8x16_t b); // VPADAL.U8 q0,q0
uint32x4_t vpadalq_u16(uint32x4_t a, uint16x8_t b); // VPADAL.U16 q0,q0
uint64x2_t vpadalq_u32(uint64x2_t a, uint32x4_t b); // VPADAL.U32 q0,q0
```

G.12 Folding maximum

vpmax -> takes maximum of adjacent pairs

```
int8x8_t    vpmax_s8(int8x8_t a, int8x8_t b);    // VPMAX.S8 d0,d0,d0
int16x4_t   vpmax_s16(int16x4_t a, int16x4_t b); // VPMAX.S16 d0,d0,d0
int32x2_t   vpmax_s32(int32x2_t a, int32x2_t b); // VPMAX.S32 d0,d0,d0
uint8x8_t   vpmax_u8(uint8x8_t a, uint8x8_t b);  // VPMAX.U8 d0,d0,d0
uint16x4_t  vpmax_u16(uint16x4_t a, uint16x4_t b); // VPMAX.U16 d0,d0,d0
uint32x2_t  vpmax_u32(uint32x2_t a, uint32x2_t b); // VPMAX.U32 d0,d0,d0
float32x2_t vpmax_f32(float32x2_t a, float32x2_t b); // VPMAX.F32 d0,d0,d0
```

G.13 Folding minimum

vpmin -> takes minimum of adjacent pairs

```
int8x8_t    vpmin_s8(int8x8_t a, int8x8_t b);    // VPMIN.S8 d0,d0,d0
int16x4_t   vpmin_s16(int16x4_t a, int16x4_t b); // VPMIN.S16 d0,d0,d0
int32x2_t   vpmin_s32(int32x2_t a, int32x2_t b); // VPMIN.S32 d0,d0,d0
uint8x8_t   vpmin_u8(uint8x8_t a, uint8x8_t b);  // VPMIN.U8 d0,d0,d0
uint16x4_t  vpmin_u16(uint16x4_t a, uint16x4_t b); // VPMIN.U16 d0,d0,d0
uint32x2_t  vpmin_u32(uint32x2_t a, uint32x2_t b); // VPMIN.U32 d0,d0,d0
float32x2_t vpmin_f32(float32x2_t a, float32x2_t b); // VPMIN.F32 d0,d0,d0
```

G.14 Reciprocal/Sqrt

These intrinsics perform the first of two steps in an iteration of the Newton-Raphson method to converge to a reciprocal or a square root. See *VRECPS* and *VRSQRTS* on page 4-78 in the *Assembler Reference*.

```
float32x2_t vrecps_f32(float32x2_t a, float32x2_t b); // VRECPS.F32 d0, d0, d0
float32x4_t vrecpsq_f32(float32x4_t a, float32x4_t b); // VRECPS.F32 q0, q0, q0
float32x2_t vrsqrts_f32(float32x2_t a, float32x2_t b); // VRSQRTS.F32 d0, d0, d0
float32x4_t vrsqrtsq_f32(float32x4_t a, float32x4_t b); // VRSQRTS.F32 q0, q0, q0
```


G.15 Shifts by signed variable

These intrinsics provide operations including shift by signed variable.

G.15.1 Vector shift left: `vshl{q}_<type>. Vr[i] := Va[i] << Vb[i]` (negative values shift right)

```
int8x8_t   vshl_s8(int8x8_t a, int8x8_t b);    // VSHL.S8 d0,d0,d0
int16x4_t  vshl_s16(int16x4_t a, int16x4_t b); // VSHL.S16 d0,d0,d0
int32x2_t  vshl_s32(int32x2_t a, int32x2_t b); // VSHL.S32 d0,d0,d0
int64x1_t  vshl_s64(int64x1_t a, int64x1_t b); // VSHL.S64 d0,d0,d0
uint8x8_t  vshl_u8(uint8x8_t a, int8x8_t b);   // VSHL.U8 d0,d0,d0
uint16x4_t vshl_u16(uint16x4_t a, int16x4_t b); // VSHL.U16 d0,d0,d0
uint32x2_t vshl_u32(uint32x2_t a, int32x2_t b); // VSHL.U32 d0,d0,d0
uint64x1_t vshl_u64(uint64x1_t a, int64x1_t b); // VSHL.U64 d0,d0,d0
int8x16_t  vshlq_s8(int8x16_t a, int8x16_t b); // VSHL.S8 q0,q0,q0
int16x8_t  vshlq_s16(int16x8_t a, int16x8_t b); // VSHL.S16 q0,q0,q0
int32x4_t  vshlq_s32(int32x4_t a, int32x4_t b); // VSHL.S32 q0,q0,q0
int64x2_t  vshlq_s64(int64x2_t a, int64x2_t b); // VSHL.S64 q0,q0,q0
uint8x16_t vshlq_u8(uint8x16_t a, int8x16_t b); // VSHL.U8 q0,q0,q0
uint16x8_t vshlq_u16(uint16x8_t a, int16x8_t b); // VSHL.U16 q0,q0,q0
uint32x4_t vshlq_u32(uint32x4_t a, int32x4_t b); // VSHL.U32 q0,q0,q0
uint64x2_t vshlq_u64(uint64x2_t a, int64x2_t b); // VSHL.U64 q0,q0,q0
```

G.15.2 Vector saturating shift left: (negative values shift right)

```
int8x8_t   vqshl_s8(int8x8_t a, int8x8_t b);    // VQSHL.S8 d0,d0,d0
int16x4_t  vqshl_s16(int16x4_t a, int16x4_t b); // VQSHL.S16 d0,d0,d0
int32x2_t  vqshl_s32(int32x2_t a, int32x2_t b); // VQSHL.S32 d0,d0,d0
int64x1_t  vqshl_s64(int64x1_t a, int64x1_t b); // VQSHL.S64 d0,d0,d0
uint8x8_t  vqshl_u8(uint8x8_t a, int8x8_t b);   // VQSHL.U8 d0,d0,d0
uint16x4_t vqshl_u16(uint16x4_t a, int16x4_t b); // VQSHL.U16 d0,d0,d0
uint32x2_t vqshl_u32(uint32x2_t a, int32x2_t b); // VQSHL.U32 d0,d0,d0
uint64x1_t vqshl_u64(uint64x1_t a, int64x1_t b); // VQSHL.U64 d0,d0,d0
int8x16_t  vqshlq_s8(int8x16_t a, int8x16_t b); // VQSHL.S8 q0,q0,q0
int16x8_t  vqshlq_s16(int16x8_t a, int16x8_t b); // VQSHL.S16 q0,q0,q0
int32x4_t  vqshlq_s32(int32x4_t a, int32x4_t b); // VQSHL.S32 q0,q0,q0
int64x2_t  vqshlq_s64(int64x2_t a, int64x2_t b); // VQSHL.S64 q0,q0,q0
uint8x16_t vqshlq_u8(uint8x16_t a, int8x16_t b); // VQSHL.U8 q0,q0,q0
uint16x8_t vqshlq_u16(uint16x8_t a, int16x8_t b); // VQSHL.U16 q0,q0,q0
uint32x4_t vqshlq_u32(uint32x4_t a, int32x4_t b); // VQSHL.U32 q0,q0,q0
uint64x2_t vqshlq_u64(uint64x2_t a, int64x2_t b); // VQSHL.U64 q0,q0,q0
```

G.15.3 Vector rounding shift left: (negative values shift right)

```
int8x8_t   vrshl_s8(int8x8_t a, int8x8_t b);    // VRSHL.S8 d0,d0,d0
int16x4_t  vrshl_s16(int16x4_t a, int16x4_t b); // VRSHL.S16 d0,d0,d0
int32x2_t  vrshl_s32(int32x2_t a, int32x2_t b); // VRSHL.S32 d0,d0,d0
int64x1_t  vrshl_s64(int64x1_t a, int64x1_t b); // VRSHL.S64 d0,d0,d0
uint8x8_t  vrshl_u8(uint8x8_t a, int8x8_t b);   // VRSHL.U8 d0,d0,d0
uint16x4_t vrshl_u16(uint16x4_t a, int16x4_t b); // VRSHL.U16 d0,d0,d0
uint32x2_t vrshl_u32(uint32x2_t a, int32x2_t b); // VRSHL.U32 d0,d0,d0
uint64x1_t vrshl_u64(uint64x1_t a, int64x1_t b); // VRSHL.U64 d0,d0,d0
int8x16_t  vrshlq_s8(int8x16_t a, int8x16_t b); // VRSHL.S8 q0,q0,q0
int16x8_t  vrshlq_s16(int16x8_t a, int16x8_t b); // VRSHL.S16 q0,q0,q0
int32x4_t  vrshlq_s32(int32x4_t a, int32x4_t b); // VRSHL.S32 q0,q0,q0
int64x2_t  vrshlq_s64(int64x2_t a, int64x2_t b); // VRSHL.S64 q0,q0,q0
uint8x16_t vrshlq_u8(uint8x16_t a, int8x16_t b); // VRSHL.U8 q0,q0,q0
uint16x8_t vrshlq_u16(uint16x8_t a, int16x8_t b); // VRSHL.U16 q0,q0,q0
uint32x4_t vrshlq_u32(uint32x4_t a, int32x4_t b); // VRSHL.U32 q0,q0,q0
uint64x2_t vrshlq_u64(uint64x2_t a, int64x2_t b); // VRSHL.U64 q0,q0,q0
```

G.15.4 Vector saturating rounding shift left: (negative values shift right)

```

int8x8_t   vqrshl_s8(int8x8_t a, int8x8_t b);    // VQRSHL.S8 d0,d0,d0
int16x4_t  vqrshl_s16(int16x4_t a, int16x4_t b); // VQRSHL.S16 d0,d0,d0
int32x2_t  vqrshl_s32(int32x2_t a, int32x2_t b); // VQRSHL.S32 d0,d0,d0
int64x1_t  vqrshl_s64(int64x1_t a, int64x1_t b); // VQRSHL.S64 d0,d0,d0
uint8x8_t  vqrshl_u8(uint8x8_t a, int8x8_t b);   // VQRSHL.U8 d0,d0,d0
uint16x4_t vqrshl_u16(uint16x4_t a, int16x4_t b); // VQRSHL.U16 d0,d0,d0
uint32x2_t vqrshl_u32(uint32x2_t a, int32x2_t b); // VQRSHL.U32 d0,d0,d0
uint64x1_t vqrshl_u64(uint64x1_t a, int64x1_t b); // VQRSHL.U64 d0,d0,d0
int8x16_t  vqrshlq_s8(int8x16_t a, int8x16_t b); // VQRSHL.S8 q0,q0,q0
int16x8_t  vqrshlq_s16(int16x8_t a, int16x8_t b); // VQRSHL.S16 q0,q0,q0
int32x4_t  vqrshlq_s32(int32x4_t a, int32x4_t b); // VQRSHL.S32 q0,q0,q0
int64x2_t  vqrshlq_s64(int64x2_t a, int64x2_t b); // VQRSHL.S64 q0,q0,q0
uint8x16_t vqrshlq_u8(uint8x16_t a, int8x16_t b); // VQRSHL.U8 q0,q0,q0
uint16x8_t vqrshlq_u16(uint16x8_t a, int16x8_t b); // VQRSHL.U16 q0,q0,q0
uint32x4_t vqrshlq_u32(uint32x4_t a, int32x4_t b); // VQRSHL.U32 q0,q0,q0
uint64x2_t vqrshlq_u64(uint64x2_t a, int64x2_t b); // VQRSHL.U64 q0,q0,q0

```

G.16 Shifts by a constant

These intrinsics provide operations for shifting by a constant.

G.16.1 Vector shift right by constant

```
int8x8_t   vshr_n_s8(int8x8_t a, __constrange(1,8) int b);    // VSHR.S8 d0,d0,#8
int16x4_t  vshr_n_s16(int16x4_t a, __constrange(1,16) int b); // VSHR.S16 d0,d0,#16
int32x2_t  vshr_n_s32(int32x2_t a, __constrange(1,32) int b); // VSHR.S32 d0,d0,#32
int64x1_t  vshr_n_s64(int64x1_t a, __constrange(1,64) int b); // VSHR.S64 d0,d0,#64
uint8x8_t  vshr_n_u8(uint8x8_t a, __constrange(1,8) int b);   // VSHR.U8 d0,d0,#8
uint16x4_t vshr_n_u16(uint16x4_t a, __constrange(1,16) int b); // VSHR.U16 d0,d0,#16
uint32x2_t vshr_n_u32(uint32x2_t a, __constrange(1,32) int b); // VSHR.U32 d0,d0,#32
uint64x1_t vshr_n_u64(uint64x1_t a, __constrange(1,64) int b); // VSHR.U64 d0,d0,#64
int8x16_t  vshrq_n_s8(int8x16_t a, __constrange(1,8) int b); // VSHR.S8 q0,q0,#8
int16x8_t  vshrq_n_s16(int16x8_t a, __constrange(1,16) int b); // VSHR.S16 q0,q0,#16
int32x4_t  vshrq_n_s32(int32x4_t a, __constrange(1,32) int b); // VSHR.S32 q0,q0,#32
int64x2_t  vshrq_n_s64(int64x2_t a, __constrange(1,64) int b); // VSHR.S64 q0,q0,#64
uint8x16_t vshrq_n_u8(uint8x16_t a, __constrange(1,8) int b); // VSHR.U8 q0,q0,#8
uint16x8_t vshrq_n_u16(uint16x8_t a, __constrange(1,16) int b); // VSHR.U16 q0,q0,#16
uint32x4_t vshrq_n_u32(uint32x4_t a, __constrange(1,32) int b); // VSHR.U32 q0,q0,#32
uint64x2_t vshrq_n_u64(uint64x2_t a, __constrange(1,64) int b); // VSHR.U64 q0,q0,#64
```

G.16.2 Vector shift left by constant

```
int8x8_t   vshl_n_s8(int8x8_t a, __constrange(0,7) int b);    // VSHL.I8 d0,d0,#0
int16x4_t  vshl_n_s16(int16x4_t a, __constrange(0,15) int b); // VSHL.I16 d0,d0,#0
int32x2_t  vshl_n_s32(int32x2_t a, __constrange(0,31) int b); // VSHL.I32 d0,d0,#0
int64x1_t  vshl_n_s64(int64x1_t a, __constrange(0,63) int b); // VSHL.I64 d0,d0,#0
uint8x8_t  vshl_n_u8(uint8x8_t a, __constrange(0,7) int b);   // VSHL.I8 d0,d0,#0
uint16x4_t vshl_n_u16(uint16x4_t a, __constrange(0,15) int b); // VSHL.I16 d0,d0,#0
uint32x2_t vshl_n_u32(uint32x2_t a, __constrange(0,31) int b); // VSHL.I32 d0,d0,#0
uint64x1_t vshl_n_u64(uint64x1_t a, __constrange(0,63) int b); // VSHL.I64 d0,d0,#0
int8x16_t  vshlq_n_s8(int8x16_t a, __constrange(0,7) int b); // VSHL.I8 q0,q0,#0
int16x8_t  vshlq_n_s16(int16x8_t a, __constrange(0,15) int b); // VSHL.I16 q0,q0,#0
int32x4_t  vshlq_n_s32(int32x4_t a, __constrange(0,31) int b); // VSHL.I32 q0,q0,#0
int64x2_t  vshlq_n_s64(int64x2_t a, __constrange(0,63) int b); // VSHL.I64 q0,q0,#0
uint8x16_t vshlq_n_u8(uint8x16_t a, __constrange(0,7) int b); // VSHL.I8 q0,q0,#0
uint16x8_t vshlq_n_u16(uint16x8_t a, __constrange(0,15) int b); // VSHL.I16 q0,q0,#0
uint32x4_t vshlq_n_u32(uint32x4_t a, __constrange(0,31) int b); // VSHL.I32 q0,q0,#0
uint64x2_t vshlq_n_u64(uint64x2_t a, __constrange(0,63) int b); // VSHL.I64 q0,q0,#0
```

G.16.3 Vector rounding shift right by constant

```
int8x8_t   vrshr_n_s8(int8x8_t a, __constrange(1,8) int b);    // VRSHR.S8 d0,d0,#8
int16x4_t  vrshr_n_s16(int16x4_t a, __constrange(1,16) int b); // VRSHR.S16 d0,d0,#16
int32x2_t  vrshr_n_s32(int32x2_t a, __constrange(1,32) int b); // VRSHR.S32 d0,d0,#32
int64x1_t  vrshr_n_s64(int64x1_t a, __constrange(1,64) int b); // VRSHR.S64 d0,d0,#64
uint8x8_t  vrshr_n_u8(uint8x8_t a, __constrange(1,8) int b);   // VRSHR.U8 d0,d0,#8
uint16x4_t vrshr_n_u16(uint16x4_t a, __constrange(1,16) int b); // VRSHR.U16 d0,d0,#16
uint32x2_t vrshr_n_u32(uint32x2_t a, __constrange(1,32) int b); // VRSHR.U32 d0,d0,#32
uint64x1_t vrshr_n_u64(uint64x1_t a, __constrange(1,64) int b); // VRSHR.U64 d0,d0,#64
int8x16_t  vrshrq_n_s8(int8x16_t a, __constrange(1,8) int b); // VRSHR.S8 q0,q0,#8
int16x8_t  vrshrq_n_s16(int16x8_t a, __constrange(1,16) int b); // VRSHR.S16 q0,q0,#16
int32x4_t  vrshrq_n_s32(int32x4_t a, __constrange(1,32) int b); // VRSHR.S32 q0,q0,#32
int64x2_t  vrshrq_n_s64(int64x2_t a, __constrange(1,64) int b); // VRSHR.S64 q0,q0,#64
uint8x16_t vrshrq_n_u8(uint8x16_t a, __constrange(1,8) int b); // VRSHR.U8 q0,q0,#8
uint16x8_t vrshrq_n_u16(uint16x8_t a, __constrange(1,16) int b); // VRSHR.U16 q0,q0,#16
uint32x4_t vrshrq_n_u32(uint32x4_t a, __constrange(1,32) int b); // VRSHR.U32 q0,q0,#32
uint64x2_t vrshrq_n_u64(uint64x2_t a, __constrange(1,64) int b); // VRSHR.U64 q0,q0,#64
```

G.16.4 Vector shift right by constant and accumulate

```

int8x8_t  vsra_n_s8(int8x8_t a, int8x8_t b, __constrange(1,8) int c); // VSRA.S8 d0,d0,#8
int16x4_t vsra_n_s16(int16x4_t a, int16x4_t b, __constrange(1,16) int c); // VSRA.S16 d0,d0,#16
int32x2_t vsra_n_s32(int32x2_t a, int32x2_t b, __constrange(1,32) int c); // VSRA.S32 d0,d0,#32
int64x1_t vsra_n_s64(int64x1_t a, int64x1_t b, __constrange(1,64) int c); // VSRA.S64 d0,d0,#64
uint8x8_t vsra_n_u8(uint8x8_t a, uint8x8_t b, __constrange(1,8) int c); // VSRA.U8 d0,d0,#8
uint16x4_t vsra_n_u16(uint16x4_t a, uint16x4_t b, __constrange(1,16) int c); // VSRA.U16 d0,d0,#16
uint32x2_t vsra_n_u32(uint32x2_t a, uint32x2_t b, __constrange(1,32) int c); // VSRA.U32 d0,d0,#32
uint64x1_t vsra_n_u64(uint64x1_t a, uint64x1_t b, __constrange(1,64) int c); // VSRA.U64 d0,d0,#64
int8x16_t vsraq_n_s8(int8x16_t a, int8x16_t b, __constrange(1,8) int c); // VSRA.S8 q0,q0,#8
int16x8_t vsraq_n_s16(int16x8_t a, int16x8_t b, __constrange(1,16) int c); // VSRA.S16 q0,q0,#16
int32x4_t vsraq_n_s32(int32x4_t a, int32x4_t b, __constrange(1,32) int c); // VSRA.S32 q0,q0,#32
int64x2_t vsraq_n_s64(int64x2_t a, int64x2_t b, __constrange(1,64) int c); // VSRA.S64 q0,q0,#64
uint8x16_t vsraq_n_u8(uint8x16_t a, uint8x16_t b, __constrange(1,8) int c); // VSRA.U8 q0,q0,#8
uint16x8_t vsraq_n_u16(uint16x8_t a, uint16x8_t b, __constrange(1,16) int c); // VSRA.U16 q0,q0,#16
uint32x4_t vsraq_n_u32(uint32x4_t a, uint32x4_t b, __constrange(1,32) int c); // VSRA.U32 q0,q0,#32
uint64x2_t vsraq_n_u64(uint64x2_t a, uint64x2_t b, __constrange(1,64) int c); // VSRA.U64 q0,q0,#64

```

G.16.5 Vector rounding shift right by constant and accumulate

```

int8x8_t  vrsra_n_s8(int8x8_t a, int8x8_t b, __constrange(1,8) int c); // VRSRA.S8 d0,d0,#8
int16x4_t vrsra_n_s16(int16x4_t a, int16x4_t b, __constrange(1,16) int c); // VRSRA.S16 d0,d0,#16
int32x2_t vrsra_n_s32(int32x2_t a, int32x2_t b, __constrange(1,32) int c); // VRSRA.S32 d0,d0,#32
int64x1_t vrsra_n_s64(int64x1_t a, int64x1_t b, __constrange(1,64) int c); // VRSRA.S64 d0,d0,#64
uint8x8_t vrsra_n_u8(uint8x8_t a, uint8x8_t b, __constrange(1,8) int c); // VRSRA.U8 d0,d0,#8
uint16x4_t vrsra_n_u16(uint16x4_t a, uint16x4_t b, __constrange(1,16) int c); // VRSRA.U16 d0,d0,#16
uint32x2_t vrsra_n_u32(uint32x2_t a, uint32x2_t b, __constrange(1,32) int c); // VRSRA.U32 d0,d0,#32
uint64x1_t vrsra_n_u64(uint64x1_t a, uint64x1_t b, __constrange(1,64) int c); // VRSRA.U64 d0,d0,#64
int8x16_t vrsraq_n_s8(int8x16_t a, int8x16_t b, __constrange(1,8) int c); // VRSRA.S8 q0,q0,#8
int16x8_t vrsraq_n_s16(int16x8_t a, int16x8_t b, __constrange(1,16) int c); // VRSRA.S16 q0,q0,#16
int32x4_t vrsraq_n_s32(int32x4_t a, int32x4_t b, __constrange(1,32) int c); // VRSRA.S32 q0,q0,#32
int64x2_t vrsraq_n_s64(int64x2_t a, int64x2_t b, __constrange(1,64) int c); // VRSRA.S64 q0,q0,#64
uint8x16_t vrsraq_n_u8(uint8x16_t a, uint8x16_t b, __constrange(1,8) int c); // VRSRA.U8 q0,q0,#8
uint16x8_t vrsraq_n_u16(uint16x8_t a, uint16x8_t b, __constrange(1,16) int c); // VRSRA.U16 q0,q0,#16
uint32x4_t vrsraq_n_u32(uint32x4_t a, uint32x4_t b, __constrange(1,32) int c); // VRSRA.U32 q0,q0,#32
uint64x2_t vrsraq_n_u64(uint64x2_t a, uint64x2_t b, __constrange(1,64) int c); // VRSRA.U64 q0,q0,#64

```

G.16.6 Vector saturating shift left by constant

```

int8x8_t  vqshl_n_s8(int8x8_t a, __constrange(0,7) int b); // VQSHL.S8 d0,d0,#0
int16x4_t vqshl_n_s16(int16x4_t a, __constrange(0,15) int b); // VQSHL.S16 d0,d0,#0
int32x2_t vqshl_n_s32(int32x2_t a, __constrange(0,31) int b); // VQSHL.S32 d0,d0,#0
int64x1_t vqshl_n_s64(int64x1_t a, __constrange(0,63) int b); // VQSHL.S64 d0,d0,#0
uint8x8_t vqshl_n_u8(uint8x8_t a, __constrange(0,7) int b); // VQSHL.U8 d0,d0,#0
uint16x4_t vqshl_n_u16(uint16x4_t a, __constrange(0,15) int b); // VQSHL.U16 d0,d0,#0
uint32x2_t vqshl_n_u32(uint32x2_t a, __constrange(0,31) int b); // VQSHL.U32 d0,d0,#0
uint64x1_t vqshl_n_u64(uint64x1_t a, __constrange(0,63) int b); // VQSHL.U64 d0,d0,#0
int8x16_t vqshlq_n_s8(int8x16_t a, __constrange(0,7) int b); // VQSHL.S8 q0,q0,#0
int16x8_t vqshlq_n_s16(int16x8_t a, __constrange(0,15) int b); // VQSHL.S16 q0,q0,#0
int32x4_t vqshlq_n_s32(int32x4_t a, __constrange(0,31) int b); // VQSHL.S32 q0,q0,#0
int64x2_t vqshlq_n_s64(int64x2_t a, __constrange(0,63) int b); // VQSHL.S64 q0,q0,#0
uint8x16_t vqshlq_n_u8(uint8x16_t a, __constrange(0,7) int b); // VQSHL.U8 q0,q0,#0
uint16x8_t vqshlq_n_u16(uint16x8_t a, __constrange(0,15) int b); // VQSHL.U16 q0,q0,#0
uint32x4_t vqshlq_n_u32(uint32x4_t a, __constrange(0,31) int b); // VQSHL.U32 q0,q0,#0
uint64x2_t vqshlq_n_u64(uint64x2_t a, __constrange(0,63) int b); // VQSHL.U64 q0,q0,#0

```

G.16.7 Vector signed->unsigned saturating shift left by constant

```

uint8x8_t vqshlu_n_s8(int8x8_t a, __constrange(0,7) int b); // VQSHLU.S8 d0,d0,#0
uint16x4_t vqshlu_n_s16(int16x4_t a, __constrange(0,15) int b); // VQSHLU.S16 d0,d0,#0
uint32x2_t vqshlu_n_s32(int32x2_t a, __constrange(0,31) int b); // VQSHLU.S32 d0,d0,#0

```

```
uint64x1_t vqshlu_n_s64(int64x1_t a, __constrange(0,63) int b); // VQSHLU.S64 d0,d0,#0
uint8x16_t vqshluq_n_s8(int8x16_t a, __constrange(0,7) int b); // VQSHLU.S8 q0,q0,#0
uint16x8_t vqshluq_n_s16(int16x8_t a, __constrange(0,15) int b); // VQSHLU.S16 q0,q0,#0
uint32x4_t vqshluq_n_s32(int32x4_t a, __constrange(0,31) int b); // VQSHLU.S32 q0,q0,#0
uint64x2_t vqshluq_n_s64(int64x2_t a, __constrange(0,63) int b); // VQSHLU.S64 q0,q0,#0
```

G.16.8 Vector narrowing shift right by constant

```
int8x8_t vshrn_n_s16(int16x8_t a, __constrange(1,8) int b); // VSHRN.I16 d0,q0,#8
int16x4_t vshrn_n_s32(int32x4_t a, __constrange(1,16) int b); // VSHRN.I32 d0,q0,#16
int32x2_t vshrn_n_s64(int64x2_t a, __constrange(1,32) int b); // VSHRN.I64 d0,q0,#32
uint8x8_t vshrn_n_u16(uint16x8_t a, __constrange(1,8) int b); // VSHRN.I16 d0,q0,#8
uint16x4_t vshrn_n_u32(uint32x4_t a, __constrange(1,16) int b); // VSHRN.I32 d0,q0,#16
uint32x2_t vshrn_n_u64(uint64x2_t a, __constrange(1,32) int b); // VSHRN.I64 d0,q0,#32
```

G.16.9 Vector signed->unsigned narrowing saturating shift right by constant

```
uint8x8_t vqshrun_n_s16(int16x8_t a, __constrange(1,8) int b); // VQSHRUN.S16 d0,q0,#8
uint16x4_t vqshrun_n_s32(int32x4_t a, __constrange(1,16) int b); // VQSHRUN.S32 d0,q0,#16
uint32x2_t vqshrun_n_s64(int64x2_t a, __constrange(1,32) int b); // VQSHRUN.S64 d0,q0,#32
```

G.16.10 Vector signed->unsigned rounding narrowing saturating shift right by constant

```
uint8x8_t vqrshrun_n_s16(int16x8_t a, __constrange(1,8) int b); // VQRSHRUN.S16 d0,q0,#8
uint16x4_t vqrshrun_n_s32(int32x4_t a, __constrange(1,16) int b); // VQRSHRUN.S32 d0,q0,#16
uint32x2_t vqrshrun_n_s64(int64x2_t a, __constrange(1,32) int b); // VQRSHRUN.S64 d0,q0,#32
```

G.16.11 Vector narrowing saturating shift right by constant

```
int8x8_t vqshrn_n_s16(int16x8_t a, __constrange(1,8) int b); // VQSHRN.S16 d0,q0,#8
int16x4_t vqshrn_n_s32(int32x4_t a, __constrange(1,16) int b); // VQSHRN.S32 d0,q0,#16
int32x2_t vqshrn_n_s64(int64x2_t a, __constrange(1,32) int b); // VQSHRN.S64 d0,q0,#32
uint8x8_t vqshrn_n_u16(uint16x8_t a, __constrange(1,8) int b); // VQSHRN.U16 d0,q0,#8
uint16x4_t vqshrn_n_u32(uint32x4_t a, __constrange(1,16) int b); // VQSHRN.U32 d0,q0,#16
uint32x2_t vqshrn_n_u64(uint64x2_t a, __constrange(1,32) int b); // VQSHRN.U64 d0,q0,#32
```

G.16.12 Vector rounding narrowing shift right by constant

```
int8x8_t vrshrn_n_s16(int16x8_t a, __constrange(1,8) int b); // VRSHRN.I16 d0,q0,#8
int16x4_t vrshrn_n_s32(int32x4_t a, __constrange(1,16) int b); // VRSHRN.I32 d0,q0,#16
int32x2_t vrshrn_n_s64(int64x2_t a, __constrange(1,32) int b); // VRSHRN.I64 d0,q0,#32
uint8x8_t vrshrn_n_u16(uint16x8_t a, __constrange(1,8) int b); // VRSHRN.I16 d0,q0,#8
uint16x4_t vrshrn_n_u32(uint32x4_t a, __constrange(1,16) int b); // VRSHRN.I32 d0,q0,#16
uint32x2_t vrshrn_n_u64(uint64x2_t a, __constrange(1,32) int b); // VRSHRN.I64 d0,q0,#32
```

G.16.13 Vector rounding narrowing saturating shift right by constant

```
int8x8_t vqrshrn_n_s16(int16x8_t a, __constrange(1,8) int b); // VQRSHRN.S16 d0,q0,#8
int16x4_t vqrshrn_n_s32(int32x4_t a, __constrange(1,16) int b); // VQRSHRN.S32 d0,q0,#16
int32x2_t vqrshrn_n_s64(int64x2_t a, __constrange(1,32) int b); // VQRSHRN.S64 d0,q0,#32
uint8x8_t vqrshrn_n_u16(uint16x8_t a, __constrange(1,8) int b); // VQRSHRN.U16 d0,q0,#8
uint16x4_t vqrshrn_n_u32(uint32x4_t a, __constrange(1,16) int b); // VQRSHRN.U32 d0,q0,#16
uint32x2_t vqrshrn_n_u64(uint64x2_t a, __constrange(1,32) int b); // VQRSHRN.U64 d0,q0,#32
```

G.16.14 Vector widening shift left by constant

```
int16x8_t vshll_n_s8(int8x8_t a, __constrange(0,8) int b); // VSHLL.S8 q0,d0,#0
int32x4_t vshll_n_s16(int16x4_t a, __constrange(0,16) int b); // VSHLL.S16 q0,d0,#0
int64x2_t vshll_n_s32(int32x2_t a, __constrange(0,32) int b); // VSHLL.S32 q0,d0,#0
```

```
uint16x8_t vshll_n_u8(uint8x8_t a, __constrange(0,8) int b); // VSHLL.U8 q0,d0,#0  
uint32x4_t vshll_n_u16(uint16x4_t a, __constrange(0,16) int b); // VSHLL.U16 q0,d0,#0  
uint64x2_t vshll_n_u32(uint32x2_t a, __constrange(0,32) int b); // VSHLL.U32 q0,d0,#0
```

G.17 Shifts with insert

These intrinsics provide operations including shifts with insert.

G.17.1 Vector shift right and insert

```
int8x8_t  vsri_n_s8(int8x8_t a, int8x8_t b, __constrange(1,8) int c); // VSRI.8 d0,d0,#8
int16x4_t vsri_n_s16(int16x4_t a, int16x4_t b, __constrange(1,16) int c); // VSRI.16 d0,d0,#16
int32x2_t vsri_n_s32(int32x2_t a, int32x2_t b, __constrange(1,32) int c); // VSRI.32 d0,d0,#32
int64x1_t vsri_n_s64(int64x1_t a, int64x1_t b, __constrange(1,64) int c); // VSRI.64 d0,d0,#64
uint8x8_t vsri_n_u8(uint8x8_t a, uint8x8_t b, __constrange(1,8) int c); // VSRI.8 d0,d0,#8
uint16x4_t vsri_n_u16(uint16x4_t a, uint16x4_t b, __constrange(1,16) int c); // VSRI.16 d0,d0,#16
uint32x2_t vsri_n_u32(uint32x2_t a, uint32x2_t b, __constrange(1,32) int c); // VSRI.32 d0,d0,#32
uint64x1_t vsri_n_u64(uint64x1_t a, uint64x1_t b, __constrange(1,64) int c); // VSRI.64 d0,d0,#64
poly8x8_t vsri_n_p8(poly8x8_t a, poly8x8_t b, __constrange(1,8) int c); // VSRI.8 d0,d0,#8
poly16x4_t vsri_n_p16(poly16x4_t a, poly16x4_t b, __constrange(1,16) int c); // VSRI.16 d0,d0,#16
int8x16_t vsriq_n_s8(int8x16_t a, int8x16_t b, __constrange(1,8) int c); // VSRI.8 q0,q0,#8
int16x8_t vsriq_n_s16(int16x8_t a, int16x8_t b, __constrange(1,16) int c); // VSRI.16 q0,q0,#16
int32x4_t vsriq_n_s32(int32x4_t a, int32x4_t b, __constrange(1,32) int c); // VSRI.32 q0,q0,#32
int64x2_t vsriq_n_s64(int64x2_t a, int64x2_t b, __constrange(1,64) int c); // VSRI.64 q0,q0,#64
uint8x16_t vsriq_n_u8(uint8x16_t a, uint8x16_t b, __constrange(1,8) int c); // VSRI.8 q0,q0,#8
uint16x8_t vsriq_n_u16(uint16x8_t a, uint16x8_t b, __constrange(1,16) int c); // VSRI.16 q0,q0,#16
uint32x4_t vsriq_n_u32(uint32x4_t a, uint32x4_t b, __constrange(1,32) int c); // VSRI.32 q0,q0,#32
uint64x2_t vsriq_n_u64(uint64x2_t a, uint64x2_t b, __constrange(1,64) int c); // VSRI.64 q0,q0,#64
poly8x16_t vsriq_n_p8(poly8x16_t a, poly8x16_t b, __constrange(1,8) int c); // VSRI.8 q0,q0,#8
poly16x8_t vsriq_n_p16(poly16x8_t a, poly16x8_t b, __constrange(1,16) int c); // VSRI.16 q0,q0,#16
```

G.17.2 Vector shift left and insert

```
int8x8_t  vsli_n_s8(int8x8_t a, int8x8_t b, __constrange(0,7) int c); // VSLI.8 d0,d0,#0
int16x4_t vsli_n_s16(int16x4_t a, int16x4_t b, __constrange(0,15) int c); // VSLI.16 d0,d0,#0
int32x2_t vsli_n_s32(int32x2_t a, int32x2_t b, __constrange(0,31) int c); // VSLI.32 d0,d0,#0
int64x1_t vsli_n_s64(int64x1_t a, int64x1_t b, __constrange(0,63) int c); // VSLI.64 d0,d0,#0
uint8x8_t vsli_n_u8(uint8x8_t a, uint8x8_t b, __constrange(0,7) int c); // VSLI.8 d0,d0,#0
uint16x4_t vsli_n_u16(uint16x4_t a, uint16x4_t b, __constrange(0,15) int c); // VSLI.16 d0,d0,#0
uint32x2_t vsli_n_u32(uint32x2_t a, uint32x2_t b, __constrange(0,31) int c); // VSLI.32 d0,d0,#0
uint64x1_t vsli_n_u64(uint64x1_t a, uint64x1_t b, __constrange(0,63) int c); // VSLI.64 d0,d0,#0
poly8x8_t vsli_n_p8(poly8x8_t a, poly8x8_t b, __constrange(0,7) int c); // VSLI.8 d0,d0,#0
poly16x4_t vsli_n_p16(poly16x4_t a, poly16x4_t b, __constrange(0,15) int c); // VSLI.16 d0,d0,#0
int8x16_t vsliq_n_s8(int8x16_t a, int8x16_t b, __constrange(0,7) int c); // VSLI.8 q0,q0,#0
int16x8_t vsliq_n_s16(int16x8_t a, int16x8_t b, __constrange(0,15) int c); // VSLI.16 q0,q0,#0
int32x4_t vsliq_n_s32(int32x4_t a, int32x4_t b, __constrange(0,31) int c); // VSLI.32 q0,q0,#0
int64x2_t vsliq_n_s64(int64x2_t a, int64x2_t b, __constrange(0,63) int c); // VSLI.64 q0,q0,#0
uint8x16_t vsliq_n_u8(uint8x16_t a, uint8x16_t b, __constrange(0,7) int c); // VSLI.8 q0,q0,#0
uint16x8_t vsliq_n_u16(uint16x8_t a, uint16x8_t b, __constrange(0,15) int c); // VSLI.16 q0,q0,#0
uint32x4_t vsliq_n_u32(uint32x4_t a, uint32x4_t b, __constrange(0,31) int c); // VSLI.32 q0,q0,#0
uint64x2_t vsliq_n_u64(uint64x2_t a, uint64x2_t b, __constrange(0,63) int c); // VSLI.64 q0,q0,#0
poly8x16_t vsliq_n_p8(poly8x16_t a, poly8x16_t b, __constrange(0,7) int c); // VSLI.8 q0,q0,#0
poly16x8_t vsliq_n_p16(poly16x8_t a, poly16x8_t b, __constrange(0,15) int c); // VSLI.16 q0,q0,#0
```

G.18 Loads of a single vector or lane

Perform loads and stores of a single vector of some type.

G.18.1 Load a single vector from memory

```
uint8x16_t vld1q_u8(__transfersize(16) uint8_t const * ptr); // VLD1.8 {d0, d1}, [r0]
uint16x8_t vld1q_u16(__transfersize(8) uint16_t const * ptr); // VLD1.16 {d0, d1}, [r0]
uint32x4_t vld1q_u32(__transfersize(4) uint32_t const * ptr); // VLD1.32 {d0, d1}, [r0]
uint64x2_t vld1q_u64(__transfersize(2) uint64_t const * ptr); // VLD1.64 {d0, d1}, [r0]
int8x16_t vld1q_s8(__transfersize(16) int8_t const * ptr); // VLD1.8 {d0, d1}, [r0]
int16x8_t vld1q_s16(__transfersize(8) int16_t const * ptr); // VLD1.16 {d0, d1}, [r0]
int32x4_t vld1q_s32(__transfersize(4) int32_t const * ptr); // VLD1.32 {d0, d1}, [r0]
int64x2_t vld1q_s64(__transfersize(2) int64_t const * ptr); // VLD1.64 {d0, d1}, [r0]
float16x8_t vld1q_f16(__transfersize(8) __fp16 const * ptr); // VLD1.16 {d0, d1}, [r0]
float32x4_t vld1q_f32(__transfersize(4) float32_t const * ptr); // VLD1.32 {d0, d1}, [r0]
poly8x16_t vld1q_p8(__transfersize(16) poly8_t const * ptr); // VLD1.8 {d0, d1}, [r0]
poly16x8_t vld1q_p16(__transfersize(8) poly16_t const * ptr); // VLD1.16 {d0, d1}, [r0]
uint8x8_t vld1_u8(__transfersize(8) uint8_t const * ptr); // VLD1.8 {d0}, [r0]
uint16x4_t vld1_u16(__transfersize(4) uint16_t const * ptr); // VLD1.16 {d0}, [r0]
uint32x2_t vld1_u32(__transfersize(2) uint32_t const * ptr); // VLD1.32 {d0}, [r0]
uint64x1_t vld1_u64(__transfersize(1) uint64_t const * ptr); // VLD1.64 {d0}, [r0]
int8x8_t vld1_s8(__transfersize(8) int8_t const * ptr); // VLD1.8 {d0}, [r0]
int16x4_t vld1_s16(__transfersize(4) int16_t const * ptr); // VLD1.16 {d0}, [r0]
int32x2_t vld1_s32(__transfersize(2) int32_t const * ptr); // VLD1.32 {d0}, [r0]
int64x1_t vld1_s64(__transfersize(1) int64_t const * ptr); // VLD1.64 {d0}, [r0]
float16x4_t vld1_f16(__transfersize(4) __fp16 const * ptr); // VLD1.16 {d0}, [r0]
float32x2_t vld1_f32(__transfersize(2) float32_t const * ptr); // VLD1.32 {d0}, [r0]
poly8x8_t vld1_p8(__transfersize(8) poly8_t const * ptr); // VLD1.8 {d0}, [r0]
poly16x4_t vld1_p16(__transfersize(4) poly16_t const * ptr); // VLD1.16 {d0}, [r0]
```

G.18.2 Load a single lane from memory

```
uint8x16_t vld1q_lane_u8(__transfersize(1) uint8_t const * ptr, uint8x16_t vec, __constrange(0,15) int lane); //
// VLD1.8 {d0[0]}, [r0]

uint16x8_t vld1q_lane_u16(__transfersize(1) uint16_t const * ptr, uint16x8_t vec, __constrange(0,7) int lane);
// VLD1.16 {d0[0]}, [r0]

uint32x4_t vld1q_lane_u32(__transfersize(1) uint32_t const * ptr, uint32x4_t vec, __constrange(0,3) int lane);
// VLD1.32 {d0[0]}, [r0]

uint64x2_t vld1q_lane_u64(__transfersize(1) uint64_t const * ptr, uint64x2_t vec, __constrange(0,1) int lane);
// VLD1.64 {d0}, [r0]

int8x16_t vld1q_lane_s8(__transfersize(1) int8_t const * ptr, int8x16_t vec, __constrange(0,15) int lane); //
VLD1.8 {d0[0]}, [r0]

int16x8_t vld1q_lane_s16(__transfersize(1) int16_t const * ptr, int16x8_t vec, __constrange(0,7) int lane); //
VLD1.16 {d0[0]}, [r0]

int32x4_t vld1q_lane_s32(__transfersize(1) int32_t const * ptr, int32x4_t vec, __constrange(0,3) int lane); //
VLD1.32 {d0[0]}, [r0]

float16x8_t vld1q_lane_f16(__transfersize(1) __fp16 const * ptr, float16x8_t vec, __constrange(0,7) int lane); //
VLD1.16 {d0[0]}, [r0]

float32x4_t vld1q_lane_f32(__transfersize(1) float32_t const * ptr, float32x4_t vec, __constrange(0,3) int lane);
// VLD1.32 {d0[0]}, [r0]
```



```

int64x2_t vld1q_lane_s64(__transfersize(1) int64_t const * ptr, int64x2_t vec, __constrange(0,1) int lane); //
VLD1.64 {d0}, [r0]

poly8x16_t vld1q_lane_p8(__transfersize(1) poly8_t const * ptr, poly8x16_t vec, __constrange(0,15) int lane); //
VLD1.8 {d0[0]}, [r0]

poly16x8_t vld1q_lane_p16(__transfersize(1) poly16_t const * ptr, poly16x8_t vec, __constrange(0,7) int lane); //
// VLD1.16 {d0[0]}, [r0]

uint8x8_t vld1_lane_u8(__transfersize(1) uint8_t const * ptr, uint8x8_t vec, __constrange(0,7) int lane); //
VLD1.8 {d0[0]}, [r0]

uint16x4_t vld1_lane_u16(__transfersize(1) uint16_t const * ptr, uint16x4_t vec, __constrange(0,3) int lane); //
VLD1.16 {d0[0]}, [r0]

uint32x2_t vld1_lane_u32(__transfersize(1) uint32_t const * ptr, uint32x2_t vec, __constrange(0,1) int lane); //
VLD1.32 {d0[0]}, [r0]

uint64x1_t vld1_lane_u64(__transfersize(1) uint64_t const * ptr, uint64x1_t vec, __constrange(0,0) int lane); //
VLD1.64 {d0}, [r0]

int8x8_t vld1_lane_s8(__transfersize(1) int8_t const * ptr, int8x8_t vec, __constrange(0,7) int lane); // VLD1.8
{d0[0]}, [r0]

int16x4_t vld1_lane_s16(__transfersize(1) int16_t const * ptr, int16x4_t vec, __constrange(0,3) int lane); //
VLD1.16 {d0[0]}, [r0]

int32x2_t vld1_lane_s32(__transfersize(1) int32_t const * ptr, int32x2_t vec, __constrange(0,1) int lane); //
VLD1.32 {d0[0]}, [r0]

float16x4_t vld1q_lane_f16(__transfersize(1) __fp16 const * ptr, float16x4_t vec, __constrange(0,3) int lane); //
VLD1.16 {d0[0]}, [r0]

float32x2_t vld1_lane_f32(__transfersize(1) float32_t const * ptr, float32x2_t vec, __constrange(0,1) int lane); //
// VLD1.32 {d0[0]}, [r0]

int64x1_t vld1_lane_s64(__transfersize(1) int64_t const * ptr, int64x1_t vec, __constrange(0,0) int lane); //
VLD1.64 {d0}, [r0]

poly8x8_t vld1_lane_p8(__transfersize(1) poly8_t const * ptr, poly8x8_t vec, __constrange(0,7) int lane); //
VLD1.8 {d0[0]}, [r0]

poly16x4_t vld1_lane_p16(__transfersize(1) poly16_t const * ptr, poly16x4_t vec, __constrange(0,3) int lane); //
VLD1.16 {d0[0]}, [r0]

```

G.18.3 Load all lanes of vector with same value from memory

```

uint8x16_t vld1q_dup_u8(__transfersize(1) uint8_t const * ptr); // VLD1.8 {d0[]}, [r0]
uint16x8_t vld1q_dup_u16(__transfersize(1) uint16_t const * ptr); // VLD1.16 {d0[]}, [r0]
uint32x4_t vld1q_dup_u32(__transfersize(1) uint32_t const * ptr); // VLD1.32 {d0[]}, [r0]
uint64x2_t vld1q_dup_u64(__transfersize(1) uint64_t const * ptr); // VLD1.64 {d0}, [r0]
int8x16_t vld1q_dup_s8(__transfersize(1) int8_t const * ptr); // VLD1.8 {d0[]}, [r0]
int16x8_t vld1q_dup_s16(__transfersize(1) int16_t const * ptr); // VLD1.16 {d0[]}, [r0]
int32x4_t vld1q_dup_s32(__transfersize(1) int32_t const * ptr); // VLD1.32 {d0[]}, [r0]
int64x2_t vld1q_dup_s64(__transfersize(1) int64_t const * ptr); // VLD1.64 {d0}, [r0]
float16x8_t vld1q_dup_f16(__transfersize(1) __fp16 const * ptr); // VLD1.16 {d0[]}, [r0]
float32x4_t vld1q_dup_f32(__transfersize(1) float32_t const * ptr); // VLD1.32 {d0[]}, [r0]
poly8x16_t vld1q_dup_p8(__transfersize(1) poly8_t const * ptr); // VLD1.8 {d0[]}, [r0]
poly16x8_t vld1q_dup_p16(__transfersize(1) poly16_t const * ptr); // VLD1.16 {d0[]}, [r0]
uint8x8_t vld1_dup_u8(__transfersize(1) uint8_t const * ptr); // VLD1.8 {d0[]}, [r0]
uint16x4_t vld1_dup_u16(__transfersize(1) uint16_t const * ptr); // VLD1.16 {d0[]}, [r0]

```

```
uint32x2_t vld1_dup_u32(__transfersize(1) uint32_t const * ptr); // VLD1.32 {d0[]}, [r0]
uint64x1_t vld1_dup_u64(__transfersize(1) uint64_t const * ptr); // VLD1.64 {d0}, [r0]
int8x8_t vld1_dup_s8(__transfersize(1) int8_t const * ptr); // VLD1.8 {d0[]}, [r0]
int16x4_t vld1_dup_s16(__transfersize(1) int16_t const * ptr); // VLD1.16 {d0[]}, [r0]
int32x2_t vld1_dup_s32(__transfersize(1) int32_t const * ptr); // VLD1.32 {d0[]}, [r0]
int64x1_t vld1_dup_s64(__transfersize(1) int64_t const * ptr); // VLD1.64 {d0}, [r0]
float16x4_t vld1_dup_f16(__transfersize(1) __fp16 const * ptr); // VLD1.16 {d0[]}, [r0]
float32x2_t vld1_dup_f32(__transfersize(1) float32_t const * ptr); // VLD1.32 {d0[]}, [r0]
poly8x8_t vld1_dup_p8(__transfersize(1) poly8_t const * ptr); // VLD1.8 {d0[]}, [r0]
poly16x4_t vld1_dup_p16(__transfersize(1) poly16_t const * ptr); // VLD1.16 {d0[]}, [r0]
```

G.19 Store a single vector or lane

Stores all lanes or a single lane of a vector.

G.19.1 Store a single vector into memory

```
void vst1q_u8(__transfersize(16) uint8_t * ptr, uint8x16_t val); // VST1.8 {d0, d1}, [r0]
void vst1q_u16(__transfersize(8) uint16_t * ptr, uint16x8_t val); // VST1.16 {d0, d1}, [r0]
void vst1q_u32(__transfersize(4) uint32_t * ptr, uint32x4_t val); // VST1.32 {d0, d1}, [r0]
void vst1q_u64(__transfersize(2) uint64_t * ptr, uint64x2_t val); // VST1.64 {d0, d1}, [r0]
void vst1q_s8(__transfersize(16) int8_t * ptr, int8x16_t val); // VST1.8 {d0, d1}, [r0]
void vst1q_s16(__transfersize(8) int16_t * ptr, int16x8_t val); // VST1.16 {d0, d1}, [r0]
void vst1q_s32(__transfersize(4) int32_t * ptr, int32x4_t val); // VST1.32 {d0, d1}, [r0]
void vst1q_s64(__transfersize(2) int64_t * ptr, int64x2_t val); // VST1.64 {d0, d1}, [r0]
void vst1q_f16(__transfersize(8) __fp16 * ptr, float16x8_t val); // VST1.16 {d0, d1}, [r0]
void vst1q_f32(__transfersize(4) float32_t * ptr, float32x4_t val); // VST1.32 {d0, d1}, [r0]
void vst1q_p8(__transfersize(16) poly8_t * ptr, poly8x16_t val); // VST1.8 {d0, d1}, [r0]
void vst1q_p16(__transfersize(8) poly16_t * ptr, poly16x8_t val); // VST1.16 {d0, d1}, [r0]
void vst1_u8(__transfersize(8) uint8_t * ptr, uint8x8_t val); // VST1.8 {d0}, [r0]
void vst1_u16(__transfersize(4) uint16_t * ptr, uint16x4_t val); // VST1.16 {d0}, [r0]
void vst1_u32(__transfersize(2) uint32_t * ptr, uint32x2_t val); // VST1.32 {d0}, [r0]
void vst1_u64(__transfersize(1) uint64_t * ptr, uint64x1_t val); // VST1.64 {d0}, [r0]
void vst1_s8(__transfersize(8) int8_t * ptr, int8x8_t val); // VST1.8 {d0}, [r0]
void vst1_s16(__transfersize(4) int16_t * ptr, int16x4_t val); // VST1.16 {d0}, [r0]
void vst1_s32(__transfersize(2) int32_t * ptr, int32x2_t val); // VST1.32 {d0}, [r0]
void vst1_s64(__transfersize(1) int64_t * ptr, int64x1_t val); // VST1.64 {d0}, [r0]
void vst1_f16(__transfersize(4) __fp16 * ptr, float16x4_t val); // VST1.16 {d0}, [r0]
void vst1_f32(__transfersize(2) float32_t * ptr, float32x2_t val); // VST1.32 {d0}, [r0]
void vst1_p8(__transfersize(8) poly8_t * ptr, poly8x8_t val); // VST1.8 {d0}, [r0]
void vst1_p16(__transfersize(4) poly16_t * ptr, poly16x4_t val); // VST1.16 {d0}, [r0]
```

G.19.2 Store a lane of a vector into memory

```
void vst1q_lane_u8(__transfersize(1) uint8_t * ptr, uint8x16_t val, __constrange(0,15) int lane); // VST1.8 {d0[0]}, [r0]

void vst1q_lane_u16(__transfersize(1) uint16_t * ptr, uint16x8_t val, __constrange(0,7) int lane); // VST1.16 {d0[0]}, [r0]

void vst1q_lane_u32(__transfersize(1) uint32_t * ptr, uint32x4_t val, __constrange(0,3) int lane); // VST1.32 {d0[0]}, [r0]

void vst1q_lane_u64(__transfersize(1) uint64_t * ptr, uint64x2_t val, __constrange(0,1) int lane); // VST1.64 {d0}, [r0]

void vst1q_lane_s8(__transfersize(1) int8_t * ptr, int8x16_t val, __constrange(0,15) int lane); // VST1.8 {d0[0]}, [r0]

void vst1q_lane_s16(__transfersize(1) int16_t * ptr, int16x8_t val, __constrange(0,7) int lane); // VST1.16 {d0[0]}, [r0]

void vst1q_lane_s32(__transfersize(1) int32_t * ptr, int32x4_t val, __constrange(0,3) int lane); // VST1.32 {d0[0]}, [r0]

void vst1q_lane_s64(__transfersize(1) int64_t * ptr, int64x2_t val, __constrange(0,1) int lane); // VST1.64 {d0}, [r0]

void vst1q_lane_f16(__transfersize(1) __fp16 * ptr, float16x8_t val, __constrange(0,7) int lane); // VST1.16 {d0[0]}, [r0]
```

```

void vst1q_lane_f32(__transfersize(1) float32_t * ptr, float32x4_t val, __constrange(0,3) int lane); // VST1.32
{d0[0]}, [r0]

void vst1q_lane_p8(__transfersize(1) poly8_t * ptr, poly8x16_t val, __constrange(0,15) int lane); // VST1.8
{d0[0]}, [r0]

void vst1q_lane_p16(__transfersize(1) poly16_t * ptr, poly16x8_t val, __constrange(0,7) int lane); // VST1.16
{d0[0]}, [r0]

void vst1_lane_u8(__transfersize(1) uint8_t * ptr, uint8x8_t val, __constrange(0,7) int lane); // VST1.8
{d0[0]}, [r0]

void vst1_lane_u16(__transfersize(1) uint16_t * ptr, uint16x4_t val, __constrange(0,3) int lane); // VST1.16
{d0[0]}, [r0]

void vst1_lane_u32(__transfersize(1) uint32_t * ptr, uint32x2_t val, __constrange(0,1) int lane); // VST1.32
{d0[0]}, [r0]

void vst1_lane_u64(__transfersize(1) uint64_t * ptr, uint64x1_t val, __constrange(0,0) int lane); // VST1.64
{d0}, [r0]

void vst1_lane_s8(__transfersize(1) int8_t * ptr, int8x8_t val, __constrange(0,7) int lane); // VST1.8 {d0[0]},
[r0]

void vst1_lane_s16(__transfersize(1) int16_t * ptr, int16x4_t val, __constrange(0,3) int lane); // VST1.16
{d0[0]}, [r0]

void vst1_lane_s32(__transfersize(1) int32_t * ptr, int32x2_t val, __constrange(0,1) int lane); // VST1.32
{d0[0]}, [r0]

void vst1_lane_s64(__transfersize(1) int64_t * ptr, int64x1_t val, __constrange(0,0) int lane); // VST1.64 {d0},
[r0]

void vst1_lane_f16(__transfersize(1) __fp16 * ptr, float16x4_t val, __constrange(0,3) int lane); // VST1.16
{d0[0]}, [r0]

void vst1_lane_f32(__transfersize(1) float32_t * ptr, float32x2_t val, __constrange(0,1) int lane); // VST1.32
{d0[0]}, [r0]

void vst1_lane_p8(__transfersize(1) poly8_t * ptr, poly8x8_t val, __constrange(0,7) int lane); // VST1.8
{d0[0]}, [r0]

void vst1_lane_p16(__transfersize(1) poly16_t * ptr, poly16x4_t val, __constrange(0,3) int lane); // VST1.16
{d0[0]}, [r0]

```

G.20 Loads of an N-element structure

These intrinsics load or store an n -element structure. The array structures are defined similarly, for example the `int16x4x2_t` structure is defined as:

```
struct int16x4x2_t
{
    int16x4_t val[2];
};
```

G.20.1 Load N-element structure from memory

```
uint8x16x2_t vld2q_u8(__transfersize(32) uint8_t const * ptr); // VLD2.8 {d0, d2}, [r0]
uint16x8x2_t vld2q_u16(__transfersize(16) uint16_t const * ptr); // VLD2.16 {d0, d2}, [r0]
uint32x4x2_t vld2q_u32(__transfersize(8) uint32_t const * ptr); // VLD2.32 {d0, d2}, [r0]
int8x16x2_t vld2q_s8(__transfersize(32) int8_t const * ptr); // VLD2.8 {d0, d2}, [r0]
int16x8x2_t vld2q_s16(__transfersize(16) int16_t const * ptr); // VLD2.16 {d0, d2}, [r0]
int32x4x2_t vld2q_s32(__transfersize(8) int32_t const * ptr); // VLD2.32 {d0, d2}, [r0]
float16x8x2_t vld2q_f16(__transfersize(16) __fp16 const * ptr); // VLD2.16 {d0, d2}, [r0]
float32x4x2_t vld2q_f32(__transfersize(8) float32_t const * ptr); // VLD2.32 {d0, d2}, [r0]
poly8x16x2_t vld2q_p8(__transfersize(32) poly8_t const * ptr); // VLD2.8 {d0, d2}, [r0]
poly16x8x2_t vld2q_p16(__transfersize(16) poly16_t const * ptr); // VLD2.16 {d0, d2}, [r0]
uint8x8x2_t vld2_u8(__transfersize(16) uint8_t const * ptr); // VLD2.8 {d0, d1}, [r0]
uint16x4x2_t vld2_u16(__transfersize(8) uint16_t const * ptr); // VLD2.16 {d0, d1}, [r0]
uint32x2x2_t vld2_u32(__transfersize(4) uint32_t const * ptr); // VLD2.32 {d0, d1}, [r0]
uint64x1x2_t vld2_u64(__transfersize(2) uint64_t const * ptr); // VLD1.64 {d0, d1}, [r0]
int8x8x2_t vld2_s8(__transfersize(16) int8_t const * ptr); // VLD2.8 {d0, d1}, [r0]
int16x4x2_t vld2_s16(__transfersize(8) int16_t const * ptr); // VLD2.16 {d0, d1}, [r0]
int32x2x2_t vld2_s32(__transfersize(4) int32_t const * ptr); // VLD2.32 {d0, d1}, [r0]
int64x1x2_t vld2_s64(__transfersize(2) int64_t const * ptr); // VLD1.64 {d0, d1}, [r0]
float16x4x2_t vld2_f16(__transfersize(8) __fp16 const * ptr); // VLD2.16 {d0, d1}, [r0]
float32x2x2_t vld2_f32(__transfersize(4) float32_t const * ptr); // VLD2.32 {d0, d1}, [r0]
poly8x8x2_t vld2_p8(__transfersize(16) poly8_t const * ptr); // VLD2.8 {d0, d1}, [r0]
poly16x4x2_t vld2_p16(__transfersize(8) poly16_t const * ptr); // VLD2.16 {d0, d1}, [r0]

uint8x16x3_t vld3q_u8(__transfersize(48) uint8_t const * ptr); // VLD3.8 {d0, d2, d4}, [r0]
uint16x8x3_t vld3q_u16(__transfersize(24) uint16_t const * ptr); // VLD3.16 {d0, d2, d4}, [r0]
uint32x4x3_t vld3q_u32(__transfersize(12) uint32_t const * ptr); // VLD3.32 {d0, d2, d4}, [r0]
int8x16x3_t vld3q_s8(__transfersize(48) int8_t const * ptr); // VLD3.8 {d0, d2, d4}, [r0]
int16x8x3_t vld3q_s16(__transfersize(24) int16_t const * ptr); // VLD3.16 {d0, d2, d4}, [r0]
int32x4x3_t vld3q_s32(__transfersize(12) int32_t const * ptr); // VLD3.32 {d0, d2, d4}, [r0]
float16x8x3_t vld3q_f16(__transfersize(24) __fp16 const * ptr); // VLD3.16 {d0, d2, d4}, [r0]
float32x4x3_t vld3q_f32(__transfersize(12) float32_t const * ptr); // VLD3.32 {d0, d2, d4}, [r0]
poly8x16x3_t vld3q_p8(__transfersize(48) poly8_t const * ptr); // VLD3.8 {d0, d2, d4}, [r0]
poly16x8x3_t vld3q_p16(__transfersize(24) poly16_t const * ptr); // VLD3.16 {d0, d2, d4}, [r0]
uint8x8x3_t vld3_u8(__transfersize(24) uint8_t const * ptr); // VLD3.8 {d0, d1, d2}, [r0]
uint16x4x3_t vld3_u16(__transfersize(12) uint16_t const * ptr); // VLD3.16 {d0, d1, d2}, [r0]
uint32x2x3_t vld3_u32(__transfersize(6) uint32_t const * ptr); // VLD3.32 {d0, d1, d2}, [r0]
uint64x1x3_t vld3_u64(__transfersize(3) uint64_t const * ptr); // VLD1.64 {d0, d1, d2}, [r0]
int8x8x3_t vld3_s8(__transfersize(24) int8_t const * ptr); // VLD3.8 {d0, d1, d2}, [r0]
int16x4x3_t vld3_s16(__transfersize(12) int16_t const * ptr); // VLD3.16 {d0, d1, d2}, [r0]
int32x2x3_t vld3_s32(__transfersize(6) int32_t const * ptr); // VLD3.32 {d0, d1, d2}, [r0]
int64x1x3_t vld3_s64(__transfersize(3) int64_t const * ptr); // VLD1.64 {d0, d1, d2}, [r0]
float16x4x3_t vld3_f16(__transfersize(12) __fp16 const * ptr); // VLD3.16 {d0, d1, d2}, [r0]
float32x2x3_t vld3_f32(__transfersize(6) float32_t const * ptr); // VLD3.32 {d0, d1, d2}, [r0]
poly8x8x3_t vld3_p8(__transfersize(24) poly8_t const * ptr); // VLD3.8 {d0, d1, d2}, [r0]
poly16x4x3_t vld3_p16(__transfersize(12) poly16_t const * ptr); // VLD3.16 {d0, d1, d2}, [r0]

uint8x16x4_t vld4q_u8(__transfersize(64) uint8_t const * ptr); // VLD4.8 {d0, d2, d4, d6}, [r0]
uint16x8x4_t vld4q_u16(__transfersize(32) uint16_t const * ptr); // VLD4.16 {d0, d2, d4, d6}, [r0]
uint32x4x4_t vld4q_u32(__transfersize(16) uint32_t const * ptr); // VLD4.32 {d0, d2, d4, d6}, [r0]
int8x16x4_t vld4q_s8(__transfersize(64) int8_t const * ptr); // VLD4.8 {d0, d2, d4, d6}, [r0]
int16x8x4_t vld4q_s16(__transfersize(32) int16_t const * ptr); // VLD4.16 {d0, d2, d4, d6}, [r0]
```

```

int32x4x4_t vld4q_s32(__transfersize(16) int32_t const * ptr); // VLD4.32 {d0, d2, d4, d6}, [r0]
float16x8x4_t vld4q_f16(__transfersize(32) __fp16 const * ptr); // VLD4.16 {d0, d2, d4, d6}, [r0]
float32x4x4_t vld4q_f32(__transfersize(16) float32_t const * ptr); // VLD4.32 {d0, d2, d4, d6}, [r0]
poly8x16x4_t vld4q_p8(__transfersize(64) poly8_t const * ptr); // VLD4.8 {d0, d2, d4, d6}, [r0]
poly16x8x4_t vld4q_p16(__transfersize(32) poly16_t const * ptr); // VLD4.16 {d0, d2, d4, d6}, [r0]
uint8x8x4_t vld4_u8(__transfersize(32) uint8_t const * ptr); // VLD4.8 {d0, d1, d2, d3}, [r0]
uint16x4x4_t vld4_u16(__transfersize(16) uint16_t const * ptr); // VLD4.16 {d0, d1, d2, d3}, [r0]
uint32x2x4_t vld4_u32(__transfersize(8) uint32_t const * ptr); // VLD4.32 {d0, d1, d2, d3}, [r0]
uint64x1x4_t vld4_u64(__transfersize(4) uint64_t const * ptr); // VLD1.64 {d0, d1, d2, d3}, [r0]
int8x8x4_t vld4_s8(__transfersize(32) int8_t const * ptr); // VLD4.8 {d0, d1, d2, d3}, [r0]
int16x4x4_t vld4_s16(__transfersize(16) int16_t const * ptr); // VLD4.16 {d0, d1, d2, d3}, [r0]
int32x2x4_t vld4_s32(__transfersize(8) int32_t const * ptr); // VLD4.32 {d0, d1, d2, d3}, [r0]
int64x1x4_t vld4_s64(__transfersize(4) int64_t const * ptr); // VLD1.64 {d0, d1, d2, d3}, [r0]
float16x4x4_t vld4_f16(__transfersize(16) __fp16 const * ptr); // VLD4.16 {d0, d1, d2, d3}, [r0]
float32x2x4_t vld4_f32(__transfersize(8) float32_t const * ptr); // VLD4.32 {d0, d1, d2, d3}, [r0]
poly8x8x4_t vld4_p8(__transfersize(32) poly8_t const * ptr); // VLD4.8 {d0, d1, d2, d3}, [r0]
poly16x4x4_t vld4_p16(__transfersize(16) poly16_t const * ptr); // VLD4.16 {d0, d1, d2, d3}, [r0]

```

G.20.2 Load all lanes of N-element structure with same value from memory

```

uint8x8x2_t vld2_dup_u8(__transfersize(2) uint8_t const * ptr); // VLD2.8 {d0[], d1[]}, [r0]
uint16x4x2_t vld2_dup_u16(__transfersize(2) uint16_t const * ptr); // VLD2.16 {d0[], d1[]}, [r0]
uint32x2x2_t vld2_dup_u32(__transfersize(2) uint32_t const * ptr); // VLD2.32 {d0[], d1[]}, [r0]
uint64x1x2_t vld2_dup_u64(__transfersize(2) uint64_t const * ptr); // VLD1.64 {d0, d1}, [r0]
int8x8x2_t vld2_dup_s8(__transfersize(2) int8_t const * ptr); // VLD2.8 {d0[], d1[]}, [r0]
int16x4x2_t vld2_dup_s16(__transfersize(2) int16_t const * ptr); // VLD2.16 {d0[], d1[]}, [r0]
int32x2x2_t vld2_dup_s32(__transfersize(2) int32_t const * ptr); // VLD2.32 {d0[], d1[]}, [r0]
int64x1x2_t vld2_dup_s64(__transfersize(2) int64_t const * ptr); // VLD1.64 {d0, d1}, [r0]
float16x4x2_t vld2_dup_f16(__transfersize(2) __fp16 const * ptr); // VLD2.16 {d0[], d1[]}, [r0]
float32x2x2_t vld2_dup_f32(__transfersize(2) float32_t const * ptr); // VLD2.32 {d0[], d1[]}, [r0]
poly8x8x2_t vld2_dup_p8(__transfersize(2) poly8_t const * ptr); // VLD2.8 {d0[], d1[]}, [r0]
poly16x4x2_t vld2_dup_p16(__transfersize(2) poly16_t const * ptr); // VLD2.16 {d0[], d1[]}, [r0]

uint8x8x3_t vld3_dup_u8(__transfersize(3) uint8_t const * ptr); // VLD3.8 {d0[], d1[], d2[]}, [r0]
uint16x4x3_t vld3_dup_u16(__transfersize(3) uint16_t const * ptr); // VLD3.16 {d0[], d1[], d2[]}, [r0]
uint32x2x3_t vld3_dup_u32(__transfersize(3) uint32_t const * ptr); // VLD3.32 {d0[], d1[], d2[]}, [r0]
uint64x1x3_t vld3_dup_u64(__transfersize(3) uint64_t const * ptr); // VLD1.64 {d0, d1, d2}, [r0]
int8x8x3_t vld3_dup_s8(__transfersize(3) int8_t const * ptr); // VLD3.8 {d0[], d1[], d2[]}, [r0]
int16x4x3_t vld3_dup_s16(__transfersize(3) int16_t const * ptr); // VLD3.16 {d0[], d1[], d2[]}, [r0]
int32x2x3_t vld3_dup_s32(__transfersize(3) int32_t const * ptr); // VLD3.32 {d0[], d1[], d2[]}, [r0]
int64x1x3_t vld3_dup_s64(__transfersize(3) int64_t const * ptr); // VLD1.64 {d0, d1, d2}, [r0]
float16x4x3_t vld3_dup_f16(__transfersize(3) __fp16 const * ptr); // VLD3.16 {d0[], d1[], d2[]}, [r0]
float32x2x3_t vld3_dup_f32(__transfersize(3) float32_t const * ptr); // VLD3.32 {d0[], d1[], d2[]}, [r0]
poly8x8x3_t vld3_dup_p8(__transfersize(3) poly8_t const * ptr); // VLD3.8 {d0[], d1[], d2[]}, [r0]
poly16x4x3_t vld3_dup_p16(__transfersize(3) poly16_t const * ptr); // VLD3.16 {d0[], d1[], d2[]}, [r0]

uint8x8x4_t vld4_dup_u8(__transfersize(4) uint8_t const * ptr); // VLD4.8 {d0[], d1[], d2[], d3[]}, [r0]
uint16x4x4_t vld4_dup_u16(__transfersize(4) uint16_t const * ptr); // VLD4.16 {d0[], d1[], d2[], d3[]}, [r0]
uint32x2x4_t vld4_dup_u32(__transfersize(4) uint32_t const * ptr); // VLD4.32 {d0[], d1[], d2[], d3[]}, [r0]
uint64x1x4_t vld4_dup_u64(__transfersize(4) uint64_t const * ptr); // VLD1.64 {d0, d1, d2, d3}, [r0]
int8x8x4_t vld4_dup_s8(__transfersize(4) int8_t const * ptr); // VLD4.8 {d0[], d1[], d2[], d3[]}, [r0]
int16x4x4_t vld4_dup_s16(__transfersize(4) int16_t const * ptr); // VLD4.16 {d0[], d1[], d2[], d3[]}, [r0]
int32x2x4_t vld4_dup_s32(__transfersize(4) int32_t const * ptr); // VLD4.32 {d0[], d1[], d2[], d3[]}, [r0]
int64x1x4_t vld4_dup_s64(__transfersize(4) int64_t const * ptr); // VLD1.64 {d0, d1, d2, d3}, [r0]
float16x4x4_t vld4_dup_f16(__transfersize(4) __fp16 const * ptr); // VLD4.16 {d0[], d1[], d2[], d3[]}, [r0]
float32x2x4_t vld4_dup_f32(__transfersize(4) float32_t const * ptr); // VLD4.32 {d0[], d1[], d2[], d3[]}, [r0]
poly8x8x4_t vld4_dup_p8(__transfersize(4) poly8_t const * ptr); // VLD4.8 {d0[], d1[], d2[], d3[]}, [r0]
poly16x4x4_t vld4_dup_p16(__transfersize(4) poly16_t const * ptr); // VLD4.16 {d0[], d1[], d2[], d3[]}, [r0]

```

G.20.3 Load a single lane of N-element structure from memory

```

uint16x8x2_t vld2q_lane_u16(__transfersize(2) uint16_t const * ptr, uint16x8x2_t src, __constrange(0,7) int
lane); // VLD2.16 {d0[0], d2[0]}, [r0]

uint32x4x2_t vld2q_lane_u32(__transfersize(2) uint32_t const * ptr, uint32x4x2_t src, __constrange(0,3) int
lane); // VLD2.32 {d0[0], d2[0]}, [r0]

int16x8x2_t vld2q_lane_s16(__transfersize(2) int16_t const * ptr, int16x8x2_t src, __constrange(0,7) int lane);
// VLD2.16 {d0[0], d2[0]}, [r0]

int32x4x2_t vld2q_lane_s32(__transfersize(2) int32_t const * ptr, int32x4x2_t src, __constrange(0,3) int lane);
// VLD2.32 {d0[0], d2[0]}, [r0]

float16x8x2_t vld2q_lane_f16(__transfersize(2) __fp16 const * ptr, float16x8x2_t src, __constrange(0,7) int
lane); // VLD2.16 {d0[0], d2[0]}, [r0]

float32x4x2_t vld2q_lane_f32(__transfersize(2) float32_t const * ptr, float32x4x2_t src, __constrange(0,3) int
lane); // VLD2.32 {d0[0], d2[0]}, [r0]

poly16x8x2_t vld2q_lane_p16(__transfersize(2) poly16_t const * ptr, poly16x8x2_t src, __constrange(0,7) int
lane); // VLD2.16 {d0[0], d2[0]}, [r0]

uint8x8x2_t vld2_lane_u8(__transfersize(2) uint8_t const * ptr, uint8x8x2_t src, __constrange(0,7) int lane); //
VLD2.8 {d0[0], d1[0]}, [r0]

uint16x4x2_t vld2_lane_u16(__transfersize(2) uint16_t const * ptr, uint16x4x2_t src, __constrange(0,3) int
lane); // VLD2.16 {d0[0], d1[0]}, [r0]

uint32x2x2_t vld2_lane_u32(__transfersize(2) uint32_t const * ptr, uint32x2x2_t src, __constrange(0,1) int
lane); // VLD2.32 {d0[0], d1[0]}, [r0]

int8x8x2_t vld2_lane_s8(__transfersize(2) int8_t const * ptr, int8x8x2_t src, __constrange(0,7) int lane); //
VLD2.8 {d0[0], d1[0]}, [r0]

int16x4x2_t vld2_lane_s16(__transfersize(2) int16_t const * ptr, int16x4x2_t src, __constrange(0,3) int lane); //
VLD2.16 {d0[0], d1[0]}, [r0]

int32x2x2_t vld2_lane_s32(__transfersize(2) int32_t const * ptr, int32x2x2_t src, __constrange(0,1) int lane); //
VLD2.32 {d0[0], d1[0]}, [r0]

float16x4x2_t vld2_lane_f16(__transfersize(2) __fp16 const * ptr, float16x4x2_t src, __constrange(0,3) int lane);
// VLD2.16 {d0[0], d1[0]}, [r0]

float32x2x2_t vld2_lane_f32(__transfersize(2) float32_t const * ptr, float32x2x2_t src, __constrange(0,1) int
lane); // VLD2.32 {d0[0], d1[0]}, [r0]

poly8x8x2_t vld2_lane_p8(__transfersize(2) poly8_t const * ptr, poly8x8x2_t src, __constrange(0,7) int lane); //
VLD2.8 {d0[0], d1[0]}, [r0]

poly16x4x2_t vld2_lane_p16(__transfersize(2) poly16_t const * ptr, poly16x4x2_t src, __constrange(0,3) int
lane); // VLD2.16 {d0[0], d1[0]}, [r0]

uint16x8x3_t vld3q_lane_u16(__transfersize(3) uint16_t const * ptr, uint16x8x3_t src, __constrange(0,7) int
lane); // VLD3.16 {d0[0], d2[0], d4[0]}, [r0]

uint32x4x3_t vld3q_lane_u32(__transfersize(3) uint32_t const * ptr, uint32x4x3_t src, __constrange(0,3) int
lane); // VLD3.32 {d0[0], d2[0], d4[0]}, [r0]

int16x8x3_t vld3q_lane_s16(__transfersize(3) int16_t const * ptr, int16x8x3_t src, __constrange(0,7) int lane);
// VLD3.16 {d0[0], d2[0], d4[0]}, [r0]

```

```

int32x4x3_t vld3q_lane_s32(__transfersize(3) int32_t const * ptr, int32x4x3_t src, __constrange(0,3) int lane);
// VLD3.32 {d0[0], d2[0], d4[0]}, [r0]

float16x8x3_t vld3q_lane_f16(__transfersize(3) __fp16 const * ptr, float16x8x3_t src, __constrange(0,7) int
lane); // VLD3.16 {d0[0], d2[0], d4[0]}, [r0]

float32x4x3_t vld3q_lane_f32(__transfersize(3) float32_t const * ptr, float32x4x3_t src, __constrange(0,3) int
lane); // VLD3.32 {d0[0], d2[0], d4[0]}, [r0]

poly16x8x3_t vld3q_lane_p16(__transfersize(3) poly16_t const * ptr, poly16x8x3_t src, __constrange(0,7) int
lane); // VLD3.16 {d0[0], d2[0], d4[0]}, [r0]

uint8x8x3_t vld3_lane_u8(__transfersize(3) uint8_t const * ptr, uint8x8x3_t src, __constrange(0,7) int lane); //
VLD3.8 {d0[0], d1[0], d2[0]}, [r0]

uint16x4x3_t vld3_lane_u16(__transfersize(3) uint16_t const * ptr, uint16x4x3_t src, __constrange(0,3) int
lane); // VLD3.16 {d0[0], d1[0], d2[0]}, [r0]

uint32x2x3_t vld3_lane_u32(__transfersize(3) uint32_t const * ptr, uint32x2x3_t src, __constrange(0,1) int
lane); // VLD3.32 {d0[0], d1[0], d2[0]}, [r0]

int8x8x3_t vld3_lane_s8(__transfersize(3) int8_t const * ptr, int8x8x3_t src, __constrange(0,7) int lane); //
VLD3.8 {d0[0], d1[0], d2[0]}, [r0]

int16x4x3_t vld3_lane_s16(__transfersize(3) int16_t const * ptr, int16x4x3_t src, __constrange(0,3) int lane); //
VLD3.16 {d0[0], d1[0], d2[0]}, [r0]

int32x2x3_t vld3_lane_s32(__transfersize(3) int32_t const * ptr, int32x2x3_t src, __constrange(0,1) int lane); //
VLD3.32 {d0[0], d1[0], d2[0]}, [r0]

float16x4x3_t vld3_lane_f16(__transfersize(3) __fp16 const * ptr, float16x4x3_t src, __constrange(0,3) int lane);
// VLD3.16 {d0[0], d1[0], d2[0]}, [r0]

float32x2x3_t vld3_lane_f32(__transfersize(3) float32_t const * ptr, float32x2x3_t src, __constrange(0,1) int
lane); // VLD3.32 {d0[0], d1[0], d2[0]}, [r0]

poly8x8x3_t vld3_lane_p8(__transfersize(3) poly8_t const * ptr, poly8x8x3_t src, __constrange(0,7) int lane); //
VLD3.8 {d0[0], d1[0], d2[0]}, [r0]

poly16x4x3_t vld3_lane_p16(__transfersize(3) poly16_t const * ptr, poly16x4x3_t src, __constrange(0,3) int
lane); // VLD3.16 {d0[0], d1[0], d2[0]}, [r0]

uint16x8x4_t vld4q_lane_u16(__transfersize(4) uint16_t const * ptr, uint16x8x4_t src, __constrange(0,7) int
lane); // VLD4.16 {d0[0], d2[0], d4[0], d6[0]}, [r0]

uint32x4x4_t vld4q_lane_u32(__transfersize(4) uint32_t const * ptr, uint32x4x4_t src, __constrange(0,3) int
lane); // VLD4.32 {d0[0], d2[0], d4[0], d6[0]}, [r0]

int16x8x4_t vld4q_lane_s16(__transfersize(4) int16_t const * ptr, int16x8x4_t src, __constrange(0,7) int lane);
// VLD4.16 {d0[0], d2[0], d4[0], d6[0]}, [r0]

int32x4x4_t vld4q_lane_s32(__transfersize(4) int32_t const * ptr, int32x4x4_t src, __constrange(0,3) int lane);
// VLD4.32 {d0[0], d2[0], d4[0], d6[0]}, [r0]

float16x8x4_t vld4q_lane_f16(__transfersize(4) __fp16 const * ptr, float16x8x4_t src, __constrange(0,7) int
lane); // VLD4.16 {d0[0], d2[0], d4[0], d6[0]}, [r0]

float32x4x4_t vld4q_lane_f32(__transfersize(4) float32_t const * ptr, float32x4x4_t src, __constrange(0,3) int
lane); // VLD4.32 {d0[0], d2[0], d4[0], d6[0]}, [r0]

poly16x8x4_t vld4q_lane_p16(__transfersize(4) poly16_t const * ptr, poly16x8x4_t src, __constrange(0,7) int
lane); // VLD4.16 {d0[0], d2[0], d4[0], d6[0]}, [r0]

```



```

uint8x8x4_t vld4_lane_u8(__transfersize(4) uint8_t const * ptr, uint8x8x4_t src, __constrange(0,7) int lane); //
VLD4.8 {d0[0], d1[0], d2[0], d3[0]}, [r0]

uint16x4x4_t vld4_lane_u16(__transfersize(4) uint16_t const * ptr, uint16x4x4_t src, __constrange(0,3) int
lane); // VLD4.16 {d0[0], d1[0], d2[0], d3[0]}, [r0]

uint32x2x4_t vld4_lane_u32(__transfersize(4) uint32_t const * ptr, uint32x2x4_t src, __constrange(0,1) int
lane); // VLD4.32 {d0[0], d1[0], d2[0], d3[0]}, [r0]

int8x8x4_t vld4_lane_s8(__transfersize(4) int8_t const * ptr, int8x8x4_t src, __constrange(0,7) int lane); //
VLD4.8 {d0[0], d1[0], d2[0], d3[0]}, [r0]

int16x4x4_t vld4_lane_s16(__transfersize(4) int16_t const * ptr, int16x4x4_t src, __constrange(0,3) int lane); //
VLD4.16 {d0[0], d1[0], d2[0], d3[0]}, [r0]

int32x2x4_t vld4_lane_s32(__transfersize(4) int32_t const * ptr, int32x2x4_t src, __constrange(0,1) int lane); //
VLD4.32 {d0[0], d1[0], d2[0], d3[0]}, [r0]

float16x4x4_t vld4_lane_f16(__transfersize(4) __fp16 const * ptr, float16x4x4_t src, __constrange(0,3) int lane);
// VLD4.16 {d0[0], d1[0], d2[0], d3[0]}, [r0]

float32x2x4_t vld4_lane_f32(__transfersize(4) float32_t const * ptr, float32x2x4_t src, __constrange(0,1) int
lane); // VLD4.32 {d0[0], d1[0], d2[0], d3[0]}, [r0]

poly8x8x4_t vld4_lane_p8(__transfersize(4) poly8_t const * ptr, poly8x8x4_t src, __constrange(0,7) int lane); //
VLD4.8 {d0[0], d1[0], d2[0], d3[0]}, [r0]

poly16x4x4_t vld4_lane_p16(__transfersize(4) poly16_t const * ptr, poly16x4x4_t src, __constrange(0,3) int
lane); // VLD4.16 {d0[0], d1[0], d2[0], d3[0]}, [r0]

```

G.20.4 Store N-element structure to memory

```

void vst2q_u8(__transfersize(32) uint8_t * ptr, uint8x16x2_t val); // VST2.8 {d0, d2}, [r0]
void vst2q_u16(__transfersize(16) uint16_t * ptr, uint16x8x2_t val); // VST2.16 {d0, d2}, [r0]
void vst2q_u32(__transfersize(8) uint32_t * ptr, uint32x4x2_t val); // VST2.32 {d0, d2}, [r0]
void vst2q_s8(__transfersize(32) int8_t * ptr, int8x16x2_t val); // VST2.8 {d0, d2}, [r0]
void vst2q_s16(__transfersize(16) int16_t * ptr, int16x8x2_t val); // VST2.16 {d0, d2}, [r0]
void vst2q_s32(__transfersize(8) int32_t * ptr, int32x4x2_t val); // VST2.32 {d0, d2}, [r0]
void vst2q_f16(__transfersize(16) __fp16 * ptr, float16x8x2_t val); // VST2.16 {d0, d2}, [r0]
void vst2q_f32(__transfersize(8) float32_t * ptr, float32x4x2_t val); // VST2.32 {d0, d2}, [r0]
void vst2q_p8(__transfersize(32) poly8_t * ptr, poly8x16x2_t val); // VST2.8 {d0, d2}, [r0]
void vst2q_p16(__transfersize(16) poly16_t * ptr, poly16x8x2_t val); // VST2.16 {d0, d2}, [r0]
void vst2_u8(__transfersize(16) uint8_t * ptr, uint8x8x2_t val); // VST2.8 {d0, d1}, [r0]
void vst2_u16(__transfersize(8) uint16_t * ptr, uint16x4x2_t val); // VST2.16 {d0, d1}, [r0]
void vst2_u32(__transfersize(4) uint32_t * ptr, uint32x2x2_t val); // VST2.32 {d0, d1}, [r0]
void vst2_u64(__transfersize(2) uint64_t * ptr, uint64x1x2_t val); // VST1.64 {d0, d1}, [r0]
void vst2_s8(__transfersize(16) int8_t * ptr, int8x8x2_t val); // VST2.8 {d0, d1}, [r0]
void vst2_s16(__transfersize(8) int16_t * ptr, int16x4x2_t val); // VST2.16 {d0, d1}, [r0]
void vst2_s32(__transfersize(4) int32_t * ptr, int32x2x2_t val); // VST2.32 {d0, d1}, [r0]
void vst2_s64(__transfersize(2) int64_t * ptr, int64x1x2_t val); // VST1.64 {d0, d1}, [r0]
void vst2_f16(__transfersize(8) __fp16 * ptr, float16x4x2_t val); // VST2.16 {d0, d1}, [r0]
void vst2_f32(__transfersize(4) float32_t * ptr, float32x2x2_t val); // VST2.32 {d0, d1}, [r0]
void vst2_p8(__transfersize(16) poly8_t * ptr, poly8x8x2_t val); // VST2.8 {d0, d1}, [r0]
void vst2_p16(__transfersize(8) poly16_t * ptr, poly16x4x2_t val); // VST2.16 {d0, d1}, [r0]

void vst3q_u8(__transfersize(48) uint8_t * ptr, uint8x16x3_t val); // VST3.8 {d0, d2, d4}, [r0]
void vst3q_u16(__transfersize(24) uint16_t * ptr, uint16x8x3_t val); // VST3.16 {d0, d2, d4}, [r0]
void vst3q_u32(__transfersize(12) uint32_t * ptr, uint32x4x3_t val); // VST3.32 {d0, d2, d4}, [r0]
void vst3q_s8(__transfersize(48) int8_t * ptr, int8x16x3_t val); // VST3.8 {d0, d2, d4}, [r0]
void vst3q_s16(__transfersize(24) int16_t * ptr, int16x8x3_t val); // VST3.16 {d0, d2, d4}, [r0]
void vst3q_s32(__transfersize(12) int32_t * ptr, int32x4x3_t val); // VST3.32 {d0, d2, d4}, [r0]
void vst3q_f16(__transfersize(24) __fp16 * ptr, float16x8x3_t val); // VST3.16 {d0, d2, d4}, [r0]

```

```

void vst3q_f32(__transfersize(12) float32_t * ptr, float32x4x3_t val); // VST3.32 {d0, d2, d4}, [r0]
void vst3q_p8(__transfersize(48) poly8_t * ptr, poly8x16x3_t val); // VST3.8 {d0, d2, d4}, [r0]
void vst3q_p16(__transfersize(24) poly16_t * ptr, poly16x8x3_t val); // VST3.16 {d0, d2, d4}, [r0]
void vst3_u8(__transfersize(24) uint8_t * ptr, uint8x8x3_t val); // VST3.8 {d0, d1, d2}, [r0]
void vst3_u16(__transfersize(12) uint16_t * ptr, uint16x4x3_t val); // VST3.16 {d0, d1, d2}, [r0]
void vst3_u32(__transfersize(6) uint32_t * ptr, uint32x2x3_t val); // VST3.32 {d0, d1, d2}, [r0]
void vst3_u64(__transfersize(3) uint64_t * ptr, uint64x1x3_t val); // VST1.64 {d0, d1, d2}, [r0]
void vst3_s8(__transfersize(24) int8_t * ptr, int8x8x3_t val); // VST3.8 {d0, d1, d2}, [r0]
void vst3_s16(__transfersize(12) int16_t * ptr, int16x4x3_t val); // VST3.16 {d0, d1, d2}, [r0]
void vst3_s32(__transfersize(6) int32_t * ptr, int32x2x3_t val); // VST3.32 {d0, d1, d2}, [r0]
void vst3_s64(__transfersize(3) int64_t * ptr, int64x1x3_t val); // VST1.64 {d0, d1, d2}, [r0]
void vst3_f16(__transfersize(12) __fp16 * ptr, float16x4x3_t val); // VST3.16 {d0, d1, d2}, [r0]
void vst3_f32(__transfersize(6) float32_t * ptr, float32x2x3_t val); // VST3.32 {d0, d1, d2}, [r0]
void vst3_p8(__transfersize(24) poly8_t * ptr, poly8x8x3_t val); // VST3.8 {d0, d1, d2}, [r0]
void vst3_p16(__transfersize(12) poly16_t * ptr, poly16x4x3_t val); // VST3.16 {d0, d1, d2}, [r0]

void vst4q_u8(__transfersize(64) uint8_t * ptr, uint8x16x4_t val); // VST4.8 {d0, d2, d4, d6}, [r0]
void vst4q_u16(__transfersize(32) uint16_t * ptr, uint16x8x4_t val); // VST4.16 {d0, d2, d4, d6}, [r0]
void vst4q_u32(__transfersize(16) uint32_t * ptr, uint32x4x4_t val); // VST4.32 {d0, d2, d4, d6}, [r0]
void vst4q_s8(__transfersize(64) int8_t * ptr, int8x16x4_t val); // VST4.8 {d0, d2, d4, d6}, [r0]
void vst4q_s16(__transfersize(32) int16_t * ptr, int16x8x4_t val); // VST4.16 {d0, d2, d4, d6}, [r0]
void vst4q_s32(__transfersize(16) int32_t * ptr, int32x4x4_t val); // VST4.32 {d0, d2, d4, d6}, [r0]
void vst4q_f16(__transfersize(32) __fp16 * ptr, float16x8x4_t val); // VST4.16 {d0, d2, d4, d6}, [r0]
void vst4q_f32(__transfersize(16) float32_t * ptr, float32x4x4_t val); // VST4.32 {d0, d2, d4, d6}, [r0]
void vst4q_p8(__transfersize(64) poly8_t * ptr, poly8x16x4_t val); // VST4.8 {d0, d2, d4, d6}, [r0]
void vst4q_p16(__transfersize(32) poly16_t * ptr, poly16x8x4_t val); // VST4.16 {d0, d2, d4, d6}, [r0]
void vst4_u8(__transfersize(32) uint8_t * ptr, uint8x8x4_t val); // VST4.8 {d0, d1, d2, d3}, [r0]
void vst4_u16(__transfersize(16) uint16_t * ptr, uint16x4x4_t val); // VST4.16 {d0, d1, d2, d3}, [r0]
void vst4_u32(__transfersize(8) uint32_t * ptr, uint32x2x4_t val); // VST4.32 {d0, d1, d2, d3}, [r0]
void vst4_u64(__transfersize(4) uint64_t * ptr, uint64x1x4_t val); // VST1.64 {d0, d1, d2, d3}, [r0]
void vst4_s8(__transfersize(32) int8_t * ptr, int8x8x4_t val); // VST4.8 {d0, d1, d2, d3}, [r0]
void vst4_s16(__transfersize(16) int16_t * ptr, int16x4x4_t val); // VST4.16 {d0, d1, d2, d3}, [r0]
void vst4_s32(__transfersize(8) int32_t * ptr, int32x2x4_t val); // VST4.32 {d0, d1, d2, d3}, [r0]
void vst4_s64(__transfersize(4) int64_t * ptr, int64x1x4_t val); // VST1.64 {d0, d1, d2, d3}, [r0]
void vst4_f16(__transfersize(16) __fp16 * ptr, float16x4x4_t val); // VST4.16 {d0, d1, d2, d3}, [r0]
void vst4_f32(__transfersize(8) float32_t * ptr, float32x2x4_t val); // VST4.32 {d0, d1, d2, d3}, [r0]
void vst4_p8(__transfersize(32) poly8_t * ptr, poly8x8x4_t val); // VST4.8 {d0, d1, d2, d3}, [r0]
void vst4_p16(__transfersize(16) poly16_t * ptr, poly16x4x4_t val); // VST4.16 {d0, d1, d2, d3}, [r0]

```

G.20.5 Store a single lane of N-element structure to memory

```

void vst2q_lane_u16(__transfersize(2) uint16_t * ptr, uint16x8x2_t val, __constrange(0,7) int lane); // VST2.16 {d0[0], d2[0]}, [r0]

void vst2q_lane_u32(__transfersize(2) uint32_t * ptr, uint32x4x2_t val, __constrange(0,3) int lane); // VST2.32 {d0[0], d2[0]}, [r0]

void vst2q_lane_s16(__transfersize(2) int16_t * ptr, int16x8x2_t val, __constrange(0,7) int lane); // VST2.16 {d0[0], d2[0]}, [r0]

void vst2q_lane_s32(__transfersize(2) int32_t * ptr, int32x4x2_t val, __constrange(0,3) int lane); // VST2.32 {d0[0], d2[0]}, [r0]

void vst2q_lane_f16(__transfersize(2) __fp16 * ptr, float16x8x2_t val, __constrange(0,7) int lane); // VST2.16 {d0[0], d2[0]}, [r0]

void vst2q_lane_f32(__transfersize(2) float32_t * ptr, float32x4x2_t val, __constrange(0,3) int lane); // VST2.32 {d0[0], d2[0]}, [r0]

void vst2q_lane_p16(__transfersize(2) poly16_t * ptr, poly16x8x2_t val, __constrange(0,7) int lane); // VST2.16 {d0[0], d2[0]}, [r0]

```

```

void vst2_lane_u8(__transfersize(2) uint8_t * ptr, uint8x8x2_t val, __constrange(0,7) int lane); // VST2.8
{d0[0], d1[0]}, [r0]

void vst2_lane_u16(__transfersize(2) uint16_t * ptr, uint16x4x2_t val, __constrange(0,3) int lane); // VST2.16
{d0[0], d1[0]}, [r0]

void vst2_lane_u32(__transfersize(2) uint32_t * ptr, uint32x2x2_t val, __constrange(0,1) int lane); // VST2.32
{d0[0], d1[0]}, [r0]

void vst2_lane_s8(__transfersize(2) int8_t * ptr, int8x8x2_t val, __constrange(0,7) int lane); // VST2.8 {d0[0],
d1[0]}, [r0]

void vst2_lane_s16(__transfersize(2) int16_t * ptr, int16x4x2_t val, __constrange(0,3) int lane); // VST2.16
{d0[0], d1[0]}, [r0]

void vst2_lane_s32(__transfersize(2) int32_t * ptr, int32x2x2_t val, __constrange(0,1) int lane); // VST2.32
{d0[0], d1[0]}, [r0]

void vst2_lane_f16(__transfersize(2) __fp16 * ptr, float16x4x2_t val, __constrange(0,3) int lane); // VST2.16
{d0[0], d1[0]}, [r0]

void vst2_lane_f32(__transfersize(2) float32_t * ptr, float32x2x2_t val, __constrange(0,1) int lane); // VST2.32
{d0[0], d1[0]}, [r0]

void vst2_lane_p8(__transfersize(2) poly8_t * ptr, poly8x8x2_t val, __constrange(0,7) int lane); // VST2.8
{d0[0], d1[0]}, [r0]

void vst2_lane_p16(__transfersize(2) poly16_t * ptr, poly16x4x2_t val, __constrange(0,3) int lane); // VST2.16
{d0[0], d1[0]}, [r0]

void vst3q_lane_u16(__transfersize(3) uint16_t * ptr, uint16x8x3_t val, __constrange(0,7) int lane); // VST3.16
{d0[0], d2[0], d4[0]}, [r0]

void vst3q_lane_u32(__transfersize(3) uint32_t * ptr, uint32x4x3_t val, __constrange(0,3) int lane); // VST3.32
{d0[0], d2[0], d4[0]}, [r0]

void vst3q_lane_s16(__transfersize(3) int16_t * ptr, int16x8x3_t val, __constrange(0,7) int lane); // VST3.16
{d0[0], d2[0], d4[0]}, [r0]

void vst3q_lane_s32(__transfersize(3) int32_t * ptr, int32x4x3_t val, __constrange(0,3) int lane); // VST3.32
{d0[0], d2[0], d4[0]}, [r0]

void vst3q_lane_f16(__transfersize(3) __fp16 * ptr, float16x8x3_t val, __constrange(0,7) int lane); // VST3.16
{d0[0], d2[0], d4[0]}, [r0]

void vst3q_lane_f32(__transfersize(3) float32_t * ptr, float32x4x3_t val, __constrange(0,3) int lane); //
VST3.32 {d0[0], d2[0], d4[0]}, [r0]

void vst3q_lane_p16(__transfersize(3) poly16_t * ptr, poly16x8x3_t val, __constrange(0,7) int lane); // VST3.16
{d0[0], d2[0], d4[0]}, [r0]

void vst3_lane_u8(__transfersize(3) uint8_t * ptr, uint8x8x3_t val, __constrange(0,7) int lane); // VST3.8
{d0[0], d1[0], d2[0]}, [r0]

void vst3_lane_u16(__transfersize(3) uint16_t * ptr, uint16x4x3_t val, __constrange(0,3) int lane); // VST3.16
{d0[0], d1[0], d2[0]}, [r0]

void vst3_lane_u32(__transfersize(3) uint32_t * ptr, uint32x2x3_t val, __constrange(0,1) int lane); // VST3.32
{d0[0], d1[0], d2[0]}, [r0]

void vst3_lane_s8(__transfersize(3) int8_t * ptr, int8x8x3_t val, __constrange(0,7) int lane); // VST3.8 {d0[0],
d1[0], d2[0]}, [r0]

```

```

void vst3_lane_s16(__transfersize(3) int16_t * ptr, int16x4x3_t val, __constrange(0,3) int lane); // VST3.16
{d0[0], d1[0], d2[0]}, [r0]

void vst3_lane_s32(__transfersize(3) int32_t * ptr, int32x2x3_t val, __constrange(0,1) int lane); // VST3.32
{d0[0], d1[0], d2[0]}, [r0]

void vst3_lane_f16(__transfersize(3) __fp16 * ptr, float16x4x3_t val, __constrange(0,3) int lane); // VST3.16
{d0[0], d1[0], d2[0]}, [r0]

void vst3_lane_f32(__transfersize(3) float32_t * ptr, float32x2x3_t val, __constrange(0,1) int lane); // VST3.32
{d0[0], d1[0], d2[0]}, [r0]

void vst3_lane_p8(__transfersize(3) poly8_t * ptr, poly8x8x3_t val, __constrange(0,7) int lane); // VST3.8
{d0[0], d1[0], d2[0]}, [r0]

void vst3_lane_p16(__transfersize(3) poly16_t * ptr, poly16x4x3_t val, __constrange(0,3) int lane); // VST3.16
{d0[0], d1[0], d2[0]}, [r0]

void vst4q_lane_u16(__transfersize(4) uint16_t * ptr, uint16x8x4_t val, __constrange(0,7) int lane); // VST4.16
{d0[0], d2[0], d4[0], d6[0]}, [r0]

void vst4q_lane_u32(__transfersize(4) uint32_t * ptr, uint32x4x4_t val, __constrange(0,3) int lane); // VST4.32
{d0[0], d2[0], d4[0], d6[0]}, [r0]

void vst4q_lane_s16(__transfersize(4) int16_t * ptr, int16x8x4_t val, __constrange(0,7) int lane); // VST4.16
{d0[0], d2[0], d4[0], d6[0]}, [r0]

void vst4q_lane_s32(__transfersize(4) int32_t * ptr, int32x4x4_t val, __constrange(0,3) int lane); // VST4.32
{d0[0], d2[0], d4[0], d6[0]}, [r0]

void vst4q_lane_f16(__transfersize(4) __fp16 * ptr, float16x8x4_t val, __constrange(0,7) int lane); // VST4.16
{d0[0], d2[0], d4[0], d6[0]}, [r0]

void vst4q_lane_f32(__transfersize(4) float32_t * ptr, float32x4x4_t val, __constrange(0,3) int lane); //
VST4.32 {d0[0], d2[0], d4[0], d6[0]}, [r0]

void vst4q_lane_p16(__transfersize(4) poly16_t * ptr, poly16x8x4_t val, __constrange(0,7) int lane); // VST4.16
{d0[0], d2[0], d4[0], d6[0]}, [r0]

void vst4_lane_u8(__transfersize(4) uint8_t * ptr, uint8x8x4_t val, __constrange(0,7) int lane); // VST4.8
{d0[0], d1[0], d2[0], d3[0]}, [r0]

void vst4_lane_u16(__transfersize(4) uint16_t * ptr, uint16x4x4_t val, __constrange(0,3) int lane); // VST4.16
{d0[0], d1[0], d2[0], d3[0]}, [r0]

void vst4_lane_u32(__transfersize(4) uint32_t * ptr, uint32x2x4_t val, __constrange(0,1) int lane); // VST4.32
{d0[0], d1[0], d2[0], d3[0]}, [r0]

void vst4_lane_s8(__transfersize(4) int8_t * ptr, int8x8x4_t val, __constrange(0,7) int lane); // VST4.8 {d0[0],
d1[0], d2[0], d3[0]}, [r0]

void vst4_lane_s16(__transfersize(4) int16_t * ptr, int16x4x4_t val, __constrange(0,3) int lane); // VST4.16
{d0[0], d1[0], d2[0], d3[0]}, [r0]

void vst4_lane_s32(__transfersize(4) int32_t * ptr, int32x2x4_t val, __constrange(0,1) int lane); // VST4.32
{d0[0], d1[0], d2[0], d3[0]}, [r0]

void vst4_lane_f16(__transfersize(4) __fp16 * ptr, float16x4x4_t val, __constrange(0,3) int lane); // VST4.16
{d0[0], d1[0], d2[0], d3[0]}, [r0]

void vst4_lane_f32(__transfersize(4) float32_t * ptr, float32x2x4_t val, __constrange(0,1) int lane); // VST4.32
{d0[0], d1[0], d2[0], d3[0]}, [r0]

```

```
void vst4_lane_p8(__transfer_size(4) poly8_t * ptr, poly8x8x4_t val, __constrange(0,7) int lane); // VST4.8  
{d0[0], d1[0], d2[0], d3[0]}, [r0]
```

```
void vst4_lane_p16(__transfer_size(4) poly16_t * ptr, poly16x4x4_t val, __constrange(0,3) int lane); // VST4.16  
{d0[0], d1[0], d2[0], d3[0]}, [r0]
```

G.21 Extract lanes from a vector and put into a register

These intrinsics extract a single lane (element) from a vector.

```

uint8_t  vget_lane_u8(uint8x8_t vec, __constrange(0,7) int lane); // VMOV.U8 r0, d0[0]
uint16_t vget_lane_u16(uint16x4_t vec, __constrange(0,3) int lane); // VMOV.U16 r0, d0[0]
uint32_t vget_lane_u32(uint32x2_t vec, __constrange(0,1) int lane); // VMOV.32 r0, d0[0]
int8_t   vget_lane_s8(int8x8_t vec, __constrange(0,7) int lane); // VMOV.S8 r0, d0[0]
int16_t  vget_lane_s16(int16x4_t vec, __constrange(0,3) int lane); // VMOV.S16 r0, d0[0]
int32_t  vget_lane_s32(int32x2_t vec, __constrange(0,1) int lane); // VMOV.32 r0, d0[0]
poly8_t  vget_lane_p8(poly8x8_t vec, __constrange(0,7) int lane); // VMOV.U8 r0, d0[0]
poly16_t vget_lane_p16(poly16x4_t vec, __constrange(0,3) int lane); // VMOV.U16 r0, d0[0]
float32_t vget_lane_f32(float32x2_t vec, __constrange(0,1) int lane); // VMOV.32 r0, d0[0]
uint8_t  vgetq_lane_u8(uint8x16_t vec, __constrange(0,15) int lane); // VMOV.U8 r0, d0[0]
uint16_t vgetq_lane_u16(uint16x8_t vec, __constrange(0,7) int lane); // VMOV.U16 r0, d0[0]
uint32_t vgetq_lane_u32(uint32x4_t vec, __constrange(0,3) int lane); // VMOV.32 r0, d0[0]
int8_t   vgetq_lane_s8(int8x16_t vec, __constrange(0,15) int lane); // VMOV.S8 r0, d0[0]
int16_t  vgetq_lane_s16(int16x8_t vec, __constrange(0,7) int lane); // VMOV.S16 r0, d0[0]
int32_t  vgetq_lane_s32(int32x4_t vec, __constrange(0,3) int lane); // VMOV.32 r0, d0[0]
poly8_t  vgetq_lane_p8(poly8x16_t vec, __constrange(0,15) int lane); // VMOV.U8 r0, d0[0]
poly16_t vgetq_lane_p16(poly16x8_t vec, __constrange(0,7) int lane); // VMOV.U16 r0, d0[0]
float32_t vgetq_lane_f32(float32x4_t vec, __constrange(0,3) int lane); // VMOV.32 r0, d0[0]
int64_t  vget_lane_s64(int64x1_t vec, __constrange(0,0) int lane); // VMOV r0,r0,d0
uint64_t vget_lane_u64(uint64x1_t vec, __constrange(0,0) int lane); // VMOV r0,r0,d0
int64_t  vgetq_lane_s64(int64x2_t vec, __constrange(0,1) int lane); // VMOV r0,r0,d0
uint64_t vgetq_lane_u64(uint64x2_t vec, __constrange(0,1) int lane); // VMOV r0,r0,d0

```

G.22 Load a single lane of a vector from a literal

These intrinsics set a single lane (element) within a vector.

```

uint8x8_t  vset_lane_u8(uint8_t value, uint8x8_t vec, __constrange(0,7) int lane); // VMOV.8 d0[0],r0
uint16x4_t vset_lane_u16(uint16_t value, uint16x4_t vec, __constrange(0,3) int lane); // VMOV.16 d0[0],r0
uint32x2_t vset_lane_u32(uint32_t value, uint32x2_t vec, __constrange(0,1) int lane); // VMOV.32 d0[0],r0
int8x8_t   vset_lane_s8(int8_t value, int8x8_t vec, __constrange(0,7) int lane); // VMOV.8 d0[0],r0
int16x4_t  vset_lane_s16(int16_t value, int16x4_t vec, __constrange(0,3) int lane); // VMOV.16 d0[0],r0
int32x2_t  vset_lane_s32(int32_t value, int32x2_t vec, __constrange(0,1) int lane); // VMOV.32 d0[0],r0
poly8x8_t  vset_lane_p8(poly8_t value, poly8x8_t vec, __constrange(0,7) int lane); // VMOV.8 d0[0],r0
poly16x4_t vset_lane_p16(poly16_t value, poly16x4_t vec, __constrange(0,3) int lane); // VMOV.16 d0[0],r0
float32x2_t vset_lane_f32(float32_t value, float32x2_t vec, __constrange(0,1) int lane); // VMOV.32 d0[0],r0
uint8x16_t vsetq_lane_u8(uint8_t value, uint8x16_t vec, __constrange(0,15) int lane); // VMOV.8 d0[0],r0
uint16x8_t vsetq_lane_u16(uint16_t value, uint16x8_t vec, __constrange(0,7) int lane); // VMOV.16 d0[0],r0
uint32x4_t vsetq_lane_u32(uint32_t value, uint32x4_t vec, __constrange(0,3) int lane); // VMOV.32 d0[0],r0
int8x16_t  vsetq_lane_s8(int8_t value, int8x16_t vec, __constrange(0,15) int lane); // VMOV.8 d0[0],r0
int16x8_t  vsetq_lane_s16(int16_t value, int16x8_t vec, __constrange(0,7) int lane); // VMOV.16 d0[0],r0
int32x4_t  vsetq_lane_s32(int32_t value, int32x4_t vec, __constrange(0,3) int lane); // VMOV.32 d0[0],r0
poly8x16_t vsetq_lane_p8(poly8_t value, poly8x16_t vec, __constrange(0,15) int lane); // VMOV.8 d0[0],r0
poly16x8_t vsetq_lane_p16(poly16_t value, poly16x8_t vec, __constrange(0,7) int lane); // VMOV.16 d0[0],r0
float32x4_t vsetq_lane_f32(float32_t value, float32x4_t vec, __constrange(0,3) int lane); // VMOV.32 d0[0],r0
int64x1_t  vset_lane_s64(int64_t value, int64x1_t vec, __constrange(0,0) int lane); // VMOV d0,r0,r0
uint64x1_t vset_lane_u64(uint64_t value, uint64x1_t vec, __constrange(0,0) int lane); // VMOV d0,r0,r0
int64x2_t  vsetq_lane_s64(int64_t value, int64x2_t vec, __constrange(0,1) int lane); // VMOV d0,r0,r0
uint64x2_t vsetq_lane_u64(uint64_t value, uint64x2_t vec, __constrange(0,1) int lane); // VMOV d0,r0,r0

```

G.23 Initialize a vector from a literal bit pattern

These intrinsics create a vector from a literal bit pattern.

```
int8x8_t   vcreate_s8(uint64_t a); // VMOV d0,r0,r0
int16x4_t  vcreate_s16(uint64_t a); // VMOV d0,r0,r0
int32x2_t  vcreate_s32(uint64_t a); // VMOV d0,r0,r0
float16x4_t vcreate_f16(uint64_t a); // VMOV d0,r0,r0
float32x2_t vcreate_f32(uint64_t a); // VMOV d0,r0,r0
uint8x8_t  vcreate_u8(uint64_t a); // VMOV d0,r0,r0
uint16x4_t vcreate_u16(uint64_t a); // VMOV d0,r0,r0
uint32x2_t vcreate_u32(uint64_t a); // VMOV d0,r0,r0
uint64x1_t vcreate_u64(uint64_t a); // VMOV d0,r0,r0
poly8x8_t  vcreate_p8(uint64_t a); // VMOV d0,r0,r0
poly16x4_t vcreate_p16(uint64_t a); // VMOV d0,r0,r0
int64x1_t  vcreate_s64(uint64_t a); // VMOV d0,r0,r0
```


G.24 Set all lanes to same value

These intrinsics set all lanes to the same value.

G.24.1 Load all lanes of vector to the same literal value

```
uint8x8_t  vdup_n_u8(uint8_t value);      // VDUP.8 d0,r0
uint16x4_t vdup_n_u16(uint16_t value);   // VDUP.16 d0,r0
uint32x2_t vdup_n_u32(uint32_t value);   // VDUP.32 d0,r0
int8x8_t   vdup_n_s8(int8_t value);      // VDUP.8 d0,r0
int16x4_t  vdup_n_s16(int16_t value);    // VDUP.16 d0,r0
int32x2_t  vdup_n_s32(int32_t value);    // VDUP.32 d0,r0
poly8x8_t  vdup_n_p8(poly8_t value);     // VDUP.8 d0,r0
poly16x4_t vdup_n_p16(poly16_t value);   // VDUP.16 d0,r0
float32x2_t vdup_n_f32(float32_t value); // VDUP.32 d0,r0
uint8x16_t vdupq_n_u8(uint8_t value);    // VDUP.8 q0,r0
uint16x8_t vdupq_n_u16(uint16_t value);  // VDUP.16 q0,r0
uint32x4_t vdupq_n_u32(uint32_t value);  // VDUP.32 q0,r0
int8x16_t  vdupq_n_s8(int8_t value);     // VDUP.8 q0,r0
int16x8_t  vdupq_n_s16(int16_t value);   // VDUP.16 q0,r0
int32x4_t  vdupq_n_s32(int32_t value);   // VDUP.32 q0,r0
poly8x16_t vdupq_n_p8(poly8_t value);    // VDUP.8 q0,r0
poly16x8_t vdupq_n_p16(poly16_t value);  // VDUP.16 q0,r0
float32x4_t vdupq_n_f32(float32_t value); // VDUP.32 q0,r0
int64x1_t  vdup_n_s64(int64_t value);    // VMOV d0,r0,r0
uint64x1_t vdup_n_u64(uint64_t value);   // VMOV d0,r0,r0
int64x2_t  vdupq_n_s64(int64_t value);   // VMOV d0,r0,r0
uint64x2_t vdupq_n_u64(uint64_t value);  // VMOV d0,r0,r0
```

```
uint8x8_t  vmov_n_u8(uint8_t value);      // VDUP.8 d0,r0
uint16x4_t vmov_n_u16(uint16_t value);   // VDUP.16 d0,r0
uint32x2_t vmov_n_u32(uint32_t value);   // VDUP.32 d0,r0
int8x8_t   vmov_n_s8(int8_t value);      // VDUP.8 d0,r0
int16x4_t  vmov_n_s16(int16_t value);    // VDUP.16 d0,r0
int32x2_t  vmov_n_s32(int32_t value);    // VDUP.32 d0,r0
poly8x8_t  vmov_n_p8(poly8_t value);     // VDUP.8 d0,r0
poly16x4_t vmov_n_p16(poly16_t value);   // VDUP.16 d0,r0
float32x2_t vmov_n_f32(float32_t value); // VDUP.32 d0,r0
uint8x16_t vmovq_n_u8(uint8_t value);    // VDUP.8 q0,r0
uint16x8_t vmovq_n_u16(uint16_t value);  // VDUP.16 q0,r0
uint32x4_t vmovq_n_u32(uint32_t value);  // VDUP.32 q0,r0
int8x16_t  vmovq_n_s8(int8_t value);     // VDUP.8 q0,r0
int16x8_t  vmovq_n_s16(int16_t value);   // VDUP.16 q0,r0
int32x4_t  vmovq_n_s32(int32_t value);   // VDUP.32 q0,r0
poly8x16_t vmovq_n_p8(poly8_t value);    // VDUP.8 q0,r0
poly16x8_t vmovq_n_p16(poly16_t value);  // VDUP.16 q0,r0
float32x4_t vmovq_n_f32(float32_t value); // VDUP.32 q0,r0
int64x1_t  vmov_n_s64(int64_t value);    // VMOV d0,r0,r0
uint64x1_t vmov_n_u64(uint64_t value);   // VMOV d0,r0,r0
int64x2_t  vmovq_n_s64(int64_t value);   // VMOV d0,r0,r0
uint64x2_t vmovq_n_u64(uint64_t value);  // VMOV d0,r0,r0
```

G.24.2 Load all lanes of the vector to the value of a lane of a vector

```
uint8x8_t  vdup_lane_u8(uint8x8_t vec, __constrange(0,7) int lane); // VDUP.8 d0,d0[0]
uint16x4_t vdup_lane_u16(uint16x4_t vec, __constrange(0,3) int lane); // VDUP.16 d0,d0[0]
uint32x2_t vdup_lane_u32(uint32x2_t vec, __constrange(0,1) int lane); // VDUP.32 d0,d0[0]
int8x8_t   vdup_lane_s8(int8x8_t vec, __constrange(0,7) int lane); // VDUP.8 d0,d0[0]
int16x4_t  vdup_lane_s16(int16x4_t vec, __constrange(0,3) int lane); // VDUP.16 d0,d0[0]
int32x2_t  vdup_lane_s32(int32x2_t vec, __constrange(0,1) int lane); // VDUP.32 d0,d0[0]
poly8x8_t  vdup_lane_p8(poly8x8_t vec, __constrange(0,7) int lane); // VDUP.8 d0,d0[0]
poly16x4_t vdup_lane_p16(poly16x4_t vec, __constrange(0,3) int lane); // VDUP.16 d0,d0[0]
```

```

float32x2_t vdup_lane_f32(float32x2_t vec, __constrange(0,1) int lane); // VDUP.32 d0,d0[0]
uint8x16_t vdupq_lane_u8(uint8x8_t vec, __constrange(0,7) int lane); // VDUP.8 q0,d0[0]
uint16x8_t vdupq_lane_u16(uint16x4_t vec, __constrange(0,3) int lane); // VDUP.16 q0,d0[0]
uint32x4_t vdupq_lane_u32(uint32x2_t vec, __constrange(0,1) int lane); // VDUP.32 q0,d0[0]
int8x16_t vdupq_lane_s8(int8x8_t vec, __constrange(0,7) int lane); // VDUP.8 q0,d0[0]
int16x8_t vdupq_lane_s16(int16x4_t vec, __constrange(0,3) int lane); // VDUP.16 q0,d0[0]
int32x4_t vdupq_lane_s32(int32x2_t vec, __constrange(0,1) int lane); // VDUP.32 q0,d0[0]
poly8x16_t vdupq_lane_p8(poly8x8_t vec, __constrange(0,7) int lane); // VDUP.8 q0,d0[0]
poly16x8_t vdupq_lane_p16(poly16x4_t vec, __constrange(0,3) int lane); // VDUP.16 q0,d0[0]
float32x4_t vdupq_lane_f32(float32x2_t vec, __constrange(0,1) int lane); // VDUP.32 q0,d0[0]
int64x1_t vdup_lane_s64(int64x1_t vec, __constrange(0,0) int lane); // VMOV d0,d0
uint64x1_t vdup_lane_u64(uint64x1_t vec, __constrange(0,0) int lane); // VMOV d0,d0
int64x2_t vdupq_lane_s64(int64x1_t vec, __constrange(0,0) int lane); // VMOV q0,q0
uint64x2_t vdupq_lane_u64(uint64x1_t vec, __constrange(0,0) int lane); // VMOV q0,q0

```

G.25 Combining vectors

These intrinsics join two 64 bit vectors into a single 128bit vector.

```
int8x16_t  vcombine_s8(int8x8_t low, int8x8_t high);      // VMOV d0,d0
int16x8_t  vcombine_s16(int16x4_t low, int16x4_t high); // VMOV d0,d0
int32x4_t  vcombine_s32(int32x2_t low, int32x2_t high); // VMOV d0,d0
int64x2_t  vcombine_s64(int64x1_t low, int64x1_t high); // VMOV d0,d0
float16x8_t vcombine_f16(float16x4_t low, float16x4_t high); // VMOV d0,d0
float32x4_t vcombine_f32(float32x2_t low, float32x2_t high); // VMOV d0,d0
uint8x16_t vcombine_u8(uint8x8_t low, uint8x8_t high);  // VMOV d0,d0
uint16x8_t vcombine_u16(uint16x4_t low, uint16x4_t high); // VMOV d0,d0
uint32x4_t vcombine_u32(uint32x2_t low, uint32x2_t high); // VMOV d0,d0
uint64x2_t vcombine_u64(uint64x1_t low, uint64x1_t high); // VMOV d0,d0
poly8x16_t vcombine_p8(poly8x8_t low, poly8x8_t high);  // VMOV d0,d0
poly16x8_t vcombine_p16(poly16x4_t low, poly16x4_t high); // VMOV d0,d0
```

G.26 Splitting vectors

These intrinsics split a 128 bit vector into 2 component 64 bit vectors

```

int8x8_t   vget_high_s8(int8x16_t a);    // VMOV d0,d0
int16x4_t  vget_high_s16(int16x8_t a);   // VMOV d0,d0
int32x2_t  vget_high_s32(int32x4_t a);   // VMOV d0,d0
int64x1_t  vget_high_s64(int64x2_t a);   // VMOV d0,d0
float16x4_t vget_high_f16(float16x8_t a); // VMOV d0,d0
float32x2_t vget_high_f32(float32x4_t a); // VMOV d0,d0
uint8x8_t  vget_high_u8(uint8x16_t a);   // VMOV d0,d0
uint16x4_t vget_high_u16(uint16x8_t a);  // VMOV d0,d0
uint32x2_t vget_high_u32(uint32x4_t a);  // VMOV d0,d0
uint64x1_t vget_high_u64(uint64x2_t a);  // VMOV d0,d0
poly8x8_t  vget_high_p8(poly8x16_t a);   // VMOV d0,d0
poly16x4_t vget_high_p16(poly16x8_t a);  // VMOV d0,d0
int8x8_t   vget_low_s8(int8x16_t a);     // VMOV d0,d0
int16x4_t  vget_low_s16(int16x8_t a);    // VMOV d0,d0
int32x2_t  vget_low_s32(int32x4_t a);    // VMOV d0,d0
int64x1_t  vget_low_s64(int64x2_t a);    // VMOV d0,d0
float16x4_t vget_low_f16(float16x8_t a);  // VMOV d0,d0
float32x2_t vget_low_f32(float32x4_t a);  // VMOV d0,d0
uint8x8_t  vget_low_u8(uint8x16_t a);    // VMOV d0,d0
uint16x4_t vget_low_u16(uint16x8_t a);   // VMOV d0,d0
uint32x2_t vget_low_u32(uint32x4_t a);   // VMOV d0,d0
uint64x1_t vget_low_u64(uint64x2_t a);   // VMOV d0,d0
poly8x8_t  vget_low_p8(poly8x16_t a);    // VMOV d0,d0
poly16x4_t vget_low_p16(poly16x8_t a);   // VMOV d0,d0

```

G.27 Converting vectors

These intrinsics are used to convert vectors.

G.27.1 Convert from float

```
int32x2_t vcvts32_f32(float32x2_t a); // VCVT.S32.F32 d0, d0
uint32x2_t vcvtu32_f32(float32x2_t a); // VCVT.U32.F32 d0, d0
int32x4_t vcvts32_f32(float32x4_t a); // VCVT.S32.F32 q0, q0
uint32x4_t vcvtu32_f32(float32x4_t a); // VCVT.U32.F32 q0, q0
int32x2_t vcvtn_s32_f32(float32x2_t a, __constrange(1,32) int b); // VCVT.S32.F32 d0, d0, #32
uint32x2_t vcvtn_u32_f32(float32x2_t a, __constrange(1,32) int b); // VCVT.U32.F32 d0, d0, #32
int32x4_t vcvtn_s32_f32(float32x4_t a, __constrange(1,32) int b); // VCVT.S32.F32 q0, q0, #32
uint32x4_t vcvtn_u32_f32(float32x4_t a, __constrange(1,32) int b); // VCVT.U32.F32 q0, q0, #32
```

G.27.2 Convert to float

```
float32x2_t vcvtf32_s32(int32x2_t a); // VCVT.F32.S32 d0, d0
float32x2_t vcvtf32_u32(uint32x2_t a); // VCVT.F32.U32 d0, d0
float32x4_t vcvtf32_s32(int32x4_t a); // VCVT.F32.S32 q0, q0
float32x4_t vcvtf32_u32(uint32x4_t a); // VCVT.F32.U32 q0, q0
float32x2_t vcvtn_f32_s32(int32x2_t a, __constrange(1,32) int b); // VCVT.F32.S32 d0, d0, #32
float32x2_t vcvtn_f32_u32(uint32x2_t a, __constrange(1,32) int b); // VCVT.F32.U32 d0, d0, #32
float32x4_t vcvtn_f32_s32(int32x4_t a, __constrange(1,32) int b); // VCVT.F32.S32 q0, q0, #32
float32x4_t vcvtn_f32_u32(uint32x4_t a, __constrange(1,32) int b); // VCVT.F32.U32 q0, q0, #32
```

G.27.3 Convert between floats

```
float16x4_t vcvtf16_f32(float32x4_t a); // VCVT.F16.F32 d0, q0
float32x4_t vcvtf32_f16(float16x4_t a); // VCVT.F32.F16 q0, d0
```

G.27.4 Vector narrow integer

```
int8x8_t vmovns16(int16x8_t a); // VMOVN.I16 d0, q0
int16x4_t vmovns32(int32x4_t a); // VMOVN.I32 d0, q0
int32x2_t vmovns64(int64x2_t a); // VMOVN.I64 d0, q0
uint8x8_t vmovnu16(uint16x8_t a); // VMOVN.U16 d0, q0
uint16x4_t vmovnu32(uint32x4_t a); // VMOVN.U32 d0, q0
uint32x2_t vmovnu64(uint64x2_t a); // VMOVN.U64 d0, q0
```

G.27.5 Vector long move

```
int16x8_t vmovl_s8(int8x8_t a); // VMOVL.S8 q0, d0
int32x4_t vmovl_s16(int16x4_t a); // VMOVL.S16 q0, d0
int64x2_t vmovl_s32(int32x2_t a); // VMOVL.S32 q0, d0
uint16x8_t vmovl_u8(uint8x8_t a); // VMOVL.U8 q0, d0
uint32x4_t vmovl_u16(uint16x4_t a); // VMOVL.U16 q0, d0
uint64x2_t vmovl_u32(uint32x2_t a); // VMOVL.U32 q0, d0
```

G.27.6 Vector saturating narrow integer

```
int8x8_t vqmovns16(int16x8_t a); // VQMOVN.S16 d0, q0
int16x4_t vqmovns32(int32x4_t a); // VQMOVN.S32 d0, q0
int32x2_t vqmovns64(int64x2_t a); // VQMOVN.S64 d0, q0
uint8x8_t vqmovnu16(uint16x8_t a); // VQMOVN.U16 d0, q0
uint16x4_t vqmovnu32(uint32x4_t a); // VQMOVN.U32 d0, q0
uint32x2_t vqmovnu64(uint64x2_t a); // VQMOVN.U64 d0, q0
```

G.27.7 Vector saturating narrow integer signed->unsigned

```
uint8x8_t vqmovun_s16(int16x8_t a); // VQMOVUN.S16 d0,q0
uint16x4_t vqmovun_s32(int32x4_t a); // VQMOVUN.S32 d0,q0
uint32x2_t vqmovun_s64(int64x2_t a); // VQMOVUN.S64 d0,q0
```

G.28 Table look up

```

uint8x8_t vtbl1_u8(uint8x8_t a, uint8x8_t b); // VTBL.8 d0, {d0}, d0
int8x8_t vtbl1_s8(int8x8_t a, int8x8_t b); // VTBL.8 d0, {d0}, d0
poly8x8_t vtbl1_p8(poly8x8_t a, uint8x8_t b); // VTBL.8 d0, {d0}, d0
uint8x8_t vtbl2_u8(uint8x8x2_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1}, d0
int8x8_t vtbl2_s8(int8x8x2_t a, int8x8_t b); // VTBL.8 d0, {d0, d1}, d0
poly8x8_t vtbl2_p8(poly8x8x2_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1}, d0
uint8x8_t vtbl3_u8(uint8x8x3_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1, d2}, d0
int8x8_t vtbl3_s8(int8x8x3_t a, int8x8_t b); // VTBL.8 d0, {d0, d1, d2}, d0
poly8x8_t vtbl3_p8(poly8x8x3_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1, d2}, d0
uint8x8_t vtbl4_u8(uint8x8x4_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1, d2, d3}, d0
int8x8_t vtbl4_s8(int8x8x4_t a, int8x8_t b); // VTBL.8 d0, {d0, d1, d2, d3}, d0
poly8x8_t vtbl4_p8(poly8x8x4_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1, d2, d3}, d0

```

G.29 Extended table look up intrinsics

```

uint8x8_t vtbx1_u8(uint8x8_t a, uint8x8_t b, uint8x8_t c); // VTBX.8 d0, {d0}, d0
int8x8_t vtbx1_s8(int8x8_t a, int8x8_t b, int8x8_t c); // VTBX.8 d0, {d0}, d0
poly8x8_t vtbx1_p8(poly8x8_t a, poly8x8_t b, uint8x8_t c); // VTBX.8 d0, {d0}, d0
uint8x8_t vtbx2_u8(uint8x8_t a, uint8x8x2_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1}, d0
int8x8_t vtbx2_s8(int8x8_t a, int8x8x2_t b, int8x8_t c); // VTBX.8 d0, {d0, d1}, d0
poly8x8_t vtbx2_p8(poly8x8_t a, poly8x8x2_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1}, d0
uint8x8_t vtbx3_u8(uint8x8_t a, uint8x8x3_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1, d2}, d0
int8x8_t vtbx3_s8(int8x8_t a, int8x8x3_t b, int8x8_t c); // VTBX.8 d0, {d0, d1, d2}, d0
poly8x8_t vtbx3_p8(poly8x8_t a, poly8x8x3_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1, d2}, d0
uint8x8_t vtbx4_u8(uint8x8_t a, uint8x8x4_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1, d2, d3}, d0
int8x8_t vtbx4_s8(int8x8_t a, int8x8x4_t b, int8x8_t c); // VTBX.8 d0, {d0, d1, d2, d3}, d0
poly8x8_t vtbx4_p8(poly8x8_t a, poly8x8x4_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1, d2, d3}, d0

```


G.30 Operations with a scalar value

Efficient code generation for these intrinsics is only guaranteed when the scalar argument is either a constant or a use of one of the `vget_lane` intrinsics.

G.30.1 Vector multiply accumulate with scalar

```
int16x4_t  vmla_lane_s16(int16x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l); // VMLA.I16 d0, d0, d0[0]

int32x2_t  vmla_lane_s32(int32x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l); // VMLA.I32 d0, d0, d0[0]

uint16x4_t vmla_lane_u16(uint16x4_t a, uint16x4_t b, uint16x4_t v, __constrange(0,3) int l); // VMLA.I16 d0, d0, d0[0]

uint32x2_t vmla_lane_u32(uint32x2_t a, uint32x2_t b, uint32x2_t v, __constrange(0,1) int l); // VMLA.I32 d0, d0, d0[0]

float32x2_t vmla_lane_f32(float32x2_t a, float32x2_t b, float32x2_t v, __constrange(0,1) int l); // VMLA.F32 d0, d0, d0[0]

int16x8_t  vmlaq_lane_s16(int16x8_t a, int16x8_t b, int16x4_t v, __constrange(0,3) int l); // VMLA.I16 q0, q0, d0[0]

int32x4_t  vmlaq_lane_s32(int32x4_t a, int32x4_t b, int32x2_t v, __constrange(0,1) int l); // VMLA.I32 q0, q0, d0[0]

uint16x8_t vmlaq_lane_u16(uint16x8_t a, uint16x8_t b, uint16x4_t v, __constrange(0,3) int l); // VMLA.I16 q0, q0, d0[0]

uint32x4_t vmlaq_lane_u32(uint32x4_t a, uint32x4_t b, uint32x2_t v, __constrange(0,1) int l); // VMLA.I32 q0, q0, d0[0]

float32x4_t vmlaq_lane_f32(float32x4_t a, float32x4_t b, float32x2_t v, __constrange(0,1) int l); // VMLA.F32 q0, q0, d0[0]
```

G.30.2 Vector widening multiply accumulate with scalar

```
int32x4_t  vmlal_lane_s16(int32x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l); //VMLAL.S16 q0, d0, d0[0]

int64x2_t  vmlal_lane_s32(int64x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l); //VMLAL.S32 q0, d0, d0[0]

uint32x4_t vmlal_lane_u16(uint32x4_t a, uint16x4_t b, uint16x4_t v, __constrange(0,3) int l); // VMLAL.U16 q0, d0, d0[0]

uint64x2_t vmlal_lane_u32(uint64x2_t a, uint32x2_t b, uint32x2_t v, __constrange(0,1) int l); // VMLAL.U32 q0, d0, d0[0]
```

G.30.3 Vector widening saturating doubling multiply accumulate with scalar

```
int32x4_t  vqdmmlal_lane_s16(int32x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l); // VQDMLAL.S16 q0, d0, d0[0]

int64x2_t  vqdmmlal_lane_s32(int64x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l); // VQDMLAL.S32 q0, d0, d0[0]
```

G.30.4 Vector multiply subtract with scalar

```

int16x4_t  vmls_lane_s16(int16x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l); // VMLS.I16 d0, d0,
d0[0]

int32x2_t  vmls_lane_s32(int32x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l); // VMLS.I32 d0, d0,
d0[0]

uint16x4_t vmls_lane_u16(uint16x4_t a, uint16x4_t b, uint16x4_t v, __constrange(0,3) int l); // VMLS.I16 d0, d0,
d0[0]

uint32x2_t vmls_lane_u32(uint32x2_t a, uint32x2_t b, uint32x2_t v, __constrange(0,1) int l); // VMLS.I32 d0, d0,
d0[0]

float32x2_t vmls_lane_f32(float32x2_t a, float32x2_t b, float32x2_t v, __constrange(0,1) int l); // VMLS.F32 d0,
d0, d0[0]

int16x8_t  vmlsq_lane_s16(int16x8_t a, int16x8_t b, int16x4_t v, __constrange(0,3) int l); // VMLS.I16 q0, q0,
d0[0]

int32x4_t  vmlsq_lane_s32(int32x4_t a, int32x4_t b, int32x2_t v, __constrange(0,1) int l); // VMLS.I32 q0, q0,
d0[0]

uint16x8_t vmlsq_lane_u16(uint16x8_t a, uint16x8_t b, uint16x4_t v, __constrange(0,3) int l); // VMLS.I16 q0,
q0, d0[0]

uint32x4_t vmlsq_lane_u32(uint32x4_t a, uint32x4_t b, uint32x2_t v, __constrange(0,1) int l); // VMLS.I32 q0,
q0, d0[0]

float32x4_t vmlsq_lane_f32(float32x4_t a, float32x4_t b, float32x2_t v, __constrange(0,1) int l); // VMLS.F32 q0,
q0, d0[0]

```

G.30.5 Vector widening multiply subtract with scalar

```

int32x4_t  vmlsl_lane_s16(int32x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l); // VMLSL.S16 q0, d0,
d0[0]

int64x2_t  vmlsl_lane_s32(int64x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l); // VMLSL.S32 q0, d0,
d0[0]

uint32x4_t vmlsl_lane_u16(uint32x4_t a, uint16x4_t b, uint16x4_t v, __constrange(0,3) int l); // VMLSL.U16 q0,
d0, d0[0]

uint64x2_t vmlsl_lane_u32(uint64x2_t a, uint32x2_t b, uint32x2_t v, __constrange(0,1) int l); // VMLSL.U32 q0,
d0, d0[0]

```

G.30.6 Vector widening saturating doubling multiply subtract with scalar

```

int32x4_t  vqdmmlsl_lane_s16(int32x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l); // VQDMLSL.S16 q0,
d0, d0[0]

int64x2_t  vqdmmlsl_lane_s32(int64x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l); // VQDMLSL.S32 q0,
d0, d0[0]

```

G.30.7 Vector multiply by scalar

```

int16x4_t  vmul_n_s16(int16x4_t a, int16_t b); // VMUL.I16 d0,d0,d0[0]
int32x2_t  vmul_n_s32(int32x2_t a, int32_t b); // VMUL.I32 d0,d0,d0[0]
float32x2_t vmul_n_f32(float32x2_t a, float32_t b); // VMUL.F32 d0,d0,d0[0]
uint16x4_t vmul_n_u16(uint16x4_t a, uint16_t b); // VMUL.I16 d0,d0,d0[0]
uint32x2_t vmul_n_u32(uint32x2_t a, uint32_t b); // VMUL.I32 d0,d0,d0[0]
int16x8_t  vmulq_n_s16(int16x8_t a, int16_t b); // VMUL.I16 q0,q0,d0[0]

```

```
int32x4_t  vmlq_n_s32(int32x4_t a, int32_t b);    // VMUL.I32 q0,q0,d0[0]
float32x4_t vmlq_n_f32(float32x4_t a, float32_t b); // VMUL.F32 q0,q0,d0[0]
uint16x8_t  vmlq_n_u16(uint16x8_t a, uint16_t b); // VMUL.I16 q0,q0,d0[0]
uint32x4_t  vmlq_n_u32(uint32x4_t a, uint32_t b); // VMUL.I32 q0,q0,d0[0]
```

G.30.8 Vector long multiply with scalar

```
int32x4_t vmull_n_s16(int16x4_t vec1, int16_t val2); // VMULL.S16 q0,d0,d0[0]
int64x2_t vmull_n_s32(int32x2_t vec1, int32_t val2); // VMULL.S32 q0,d0,d0[0]
uint32x4_t vmull_n_u16(uint16x4_t vec1, uint16_t val2); // VMULL.U16 q0,d0,d0[0]
uint64x2_t vmull_n_u32(uint32x2_t vec1, uint32_t val2); // VMULL.U32 q0,d0,d0[0]
```

G.30.9 Vector long multiply by scalar

```
int32x4_t vmull_lane_s16(int16x4_t vec1, int16x4_t val2, __constrange(0, 3) int val3); // VMULL.S16 q0,d0,d0[0]
int64x2_t vmull_lane_s32(int32x2_t vec1, int32x2_t val2, __constrange(0, 1) int val3); // VMULL.S32 q0,d0,d0[0]
uint32x4_t vmull_lane_u16(uint16x4_t vec1, uint16x4_t val2, __constrange(0, 3) int val3); // VMULL.U16
q0,d0,d0[0]
uint64x2_t vmull_lane_u32(uint32x2_t vec1, uint32x2_t val2, __constrange(0, 1) int val3); // VMULL.U32
q0,d0,d0[0]
```

G.30.10 Vector saturating doubling long multiply with scalar

```
int32x4_t vqdmull_n_s16(int16x4_t vec1, int16_t val2); // VQDMULL.S16 q0,d0,d0[0]
int64x2_t vqdmull_n_s32(int32x2_t vec1, int32_t val2); // VQDMULL.S32 q0,d0,d0[0]
```

G.30.11 Vector saturating doubling long multiply by scalar

```
int32x4_t vqdmull_lane_s16(int16x4_t vec1, int16x4_t val2, __constrange(0, 3) int val3); // VQDMULL.S16
q0,d0,d0[0]
int64x2_t vqdmull_lane_s32(int32x2_t vec1, int32x2_t val2, __constrange(0, 1) int val3); // VQDMULL.S32
q0,d0,d0[0]
```

G.30.12 Vector saturating doubling multiply high with scalar

```
int16x4_t vqdmulh_n_s16(int16x4_t vec1, int16_t val2); // VQDMULH.S16 d0,d0,d0[0]
int32x2_t vqdmulh_n_s32(int32x2_t vec1, int32_t val2); // VQDMULH.S32 d0,d0,d0[0]
int16x8_t vqdmulhq_n_s16(int16x8_t vec1, int16_t val2); // VQDMULH.S16 q0,q0,d0[0]
int32x4_t vqdmulhq_n_s32(int32x4_t vec1, int32_t val2); // VQDMULH.S32 q0,q0,d0[0]
```

G.30.13 Vector saturating doubling multiply high by scalar

```
int16x4_t vqdmulh_lane_s16(int16x4_t vec1, int16x4_t val2, __constrange(0, 3) int val3); // VQDMULH.S16
d0,d0,d0[0]
int32x2_t vqdmulh_lane_s32(int32x2_t vec1, int32x2_t val2, __constrange(0, 1) int val3); // VQDMULH.S32
d0,d0,d0[0]
int16x8_t vqdmulhq_lane_s16(int16x8_t vec1, int16x4_t val2, __constrange(0, 3) int val3); // VQDMULH.S16
q0,q0,d0[0]
int32x4_t vqdmulhq_lane_s32(int32x4_t vec1, int32x2_t val2, __constrange(0, 1) int val3); // VQDMULH.S32
q0,q0,d0[0]
```

G.30.14 Vector saturating rounding doubling multiply high with scalar

```
int16x4_t vqrdmulh_n_s16(int16x4_t vec1, int16_t val2); // VQRDMULH.S16 d0,d0,d0[0]
int32x2_t vqrdmulh_n_s32(int32x2_t vec1, int32_t val2); // VQRDMULH.S32 d0,d0,d0[0]
int16x8_t vqrdmulhq_n_s16(int16x8_t vec1, int16_t val2); // VQRDMULH.S16 q0,q0,d0[0]
int32x4_t vqrdmulhq_n_s32(int32x4_t vec1, int32_t val2); // VQRDMULH.S32 q0,q0,d0[0]
```

G.30.15 Vector rounding saturating doubling multiply high by scalar

```
int16x4_t vqrdmulh_lane_s16(int16x4_t vec1, int16x4_t val2, __constrange(0, 3) int val3); // VQRDMULH.S16
d0,d0,d0[0]

int32x2_t vqrdmulh_lane_s32(int32x2_t vec1, int32x2_t val2, __constrange(0, 1) int val3); // VQRDMULH.S32
d0,d0,d0[0]

int16x8_t vqrdmulhq_lane_s16(int16x8_t vec1, int16x4_t val2, __constrange(0, 3) int val3); // VQRDMULH.S16
q0,q0,d0[0]

int32x4_t vqrdmulhq_lane_s32(int32x4_t vec1, int32x2_t val2, __constrange(0, 1) int val3); // VQRDMULH.S32
q0,q0,d0[0]
```

G.30.16 Vector multiply accumulate with scalar

```
int16x4_t vmla_n_s16(int16x4_t a, int16x4_t b, int16_t c); // VMLA.I16 d0, d0, d0[0]
int32x2_t vmla_n_s32(int32x2_t a, int32x2_t b, int32_t c); // VMLA.I32 d0, d0, d0[0]
uint16x4_t vmla_n_u16(uint16x4_t a, uint16x4_t b, uint16_t c); // VMLA.I16 d0, d0, d0[0]
uint32x2_t vmla_n_u32(uint32x2_t a, uint32x2_t b, uint32_t c); // VMLA.I32 d0, d0, d0[0]
float32x2_t vmla_n_f32(float32x2_t a, float32x2_t b, float32_t c); // VMLA.F32 d0, d0, d0[0]
int16x8_t vmlaq_n_s16(int16x8_t a, int16x8_t b, int16_t c); // VMLA.I16 q0, q0, d0[0]
int32x4_t vmlaq_n_s32(int32x4_t a, int32x4_t b, int32_t c); // VMLA.I32 q0, q0, d0[0]
uint16x8_t vmlaq_n_u16(uint16x8_t a, uint16x8_t b, uint16_t c); // VMLA.I16 q0, q0, d0[0]
uint32x4_t vmlaq_n_u32(uint32x4_t a, uint32x4_t b, uint32_t c); // VMLA.I32 q0, q0, d0[0]
float32x4_t vmlaq_n_f32(float32x4_t a, float32x4_t b, float32_t c); // VMLA.F32 q0, q0, d0[0]
```

G.30.17 Vector widening multiply accumulate with scalar

```
int32x4_t vmlal_n_s16(int32x4_t a, int16x4_t b, int16_t c); // VMLAL.S16 q0, d0, d0[0]
int64x2_t vmlal_n_s32(int64x2_t a, int32x2_t b, int32_t c); // VMLAL.S32 q0, d0, d0[0]
uint32x4_t vmlal_n_u16(uint32x4_t a, uint16x4_t b, uint16_t c); // VMLAL.U16 q0, d0, d0[0]
uint64x2_t vmlal_n_u32(uint64x2_t a, uint32x2_t b, uint32_t c); // VMLAL.U32 q0, d0, d0[0]
```

G.30.18 Vector widening saturating doubling multiply accumulate with scalar

```
int32x4_t vqdmmlal_n_s16(int32x4_t a, int16x4_t b, int16_t c); // VQDMLAL.S16 q0, d0, d0[0]
int64x2_t vqdmmlal_n_s32(int64x2_t a, int32x2_t b, int32_t c); // VQDMLAL.S32 q0, d0, d0[0]
```

G.30.19 Vector multiply subtract with scalar

```
int16x4_t vmls_n_s16(int16x4_t a, int16x4_t b, int16_t c); // VMLS.I16 d0, d0, d0[0]
int32x2_t vmls_n_s32(int32x2_t a, int32x2_t b, int32_t c); // VMLS.I32 d0, d0, d0[0]
uint16x4_t vmls_n_u16(uint16x4_t a, uint16x4_t b, uint16_t c); // VMLS.I16 d0, d0, d0[0]
uint32x2_t vmls_n_u32(uint32x2_t a, uint32x2_t b, uint32_t c); // VMLS.I32 d0, d0, d0[0]
float32x2_t vmls_n_f32(float32x2_t a, float32x2_t b, float32_t c); // VMLS.F32 d0, d0, d0[0]
int16x8_t vmlsq_n_s16(int16x8_t a, int16x8_t b, int16_t c); // VMLS.I16 q0, q0, d0[0]
int32x4_t vmlsq_n_s32(int32x4_t a, int32x4_t b, int32_t c); // VMLS.I32 q0, q0, d0[0]
uint16x8_t vmlsq_n_u16(uint16x8_t a, uint16x8_t b, uint16_t c); // VMLS.I16 q0, q0, d0[0]
uint32x4_t vmlsq_n_u32(uint32x4_t a, uint32x4_t b, uint32_t c); // VMLS.I32 q0, q0, d0[0]
float32x4_t vmlsq_n_f32(float32x4_t a, float32x4_t b, float32_t c); // VMLS.F32 q0, q0, d0[0]
```

G.30.20 Vector widening multiply subtract with scalar

```

int32x4_t  vmlsl_n_s16(int32x4_t a, int16x4_t b, int16_t c);    // VMLSL.S16 q0, d0, d0[0]
int64x2_t  vmlsl_n_s32(int64x2_t a, int32x2_t b, int32_t c);    // VMLSL.S32 q0, d0, d0[0]
uint32x4_t vmlsl_n_u16(uint32x4_t a, uint16x4_t b, uint16_t c); // VMLSL.U16 q0, d0, d0[0]
uint64x2_t vmlsl_n_u32(uint64x2_t a, uint32x2_t b, uint32_t c); // VMLSL.U32 q0, d0, d0[0]

```

G.30.21 Vector widening saturating doubling multiply subtract with scalar

```

int32x4_t  vqdmmlsl_n_s16(int32x4_t a, int16x4_t b, int16_t c); // VQDMLSL.S16 q0, d0, d0[0]
int64x2_t  vqdmmlsl_n_s32(int64x2_t a, int32x2_t b, int32_t c); // VQDMLSL.S32 q0, d0, d0[0]

```

G.31 Vector extract

```

int8x8_t  vext_s8(int8x8_t a, int8x8_t b, __constrange(0,7) int c);    // VEXT.8 d0,d0,d0,#0
uint8x8_t vext_u8(uint8x8_t a, uint8x8_t b, __constrange(0,7) int c); // VEXT.8 d0,d0,d0,#0
poly8x8_t vext_p8(poly8x8_t a, poly8x8_t b, __constrange(0,7) int c); // VEXT.8 d0,d0,d0,#0
int16x4_t vext_s16(int16x4_t a, int16x4_t b, __constrange(0,3) int c); // VEXT.16 d0,d0,d0,#0
uint16x4_t vext_u16(uint16x4_t a, uint16x4_t b, __constrange(0,3) int c); // VEXT.16 d0,d0,d0,#0
poly16x4_t vext_p16(poly16x4_t a, poly16x4_t b, __constrange(0,3) int c); // VEXT.16 d0,d0,d0,#0
int32x2_t vext_s32(int32x2_t a, int32x2_t b, __constrange(0,1) int c); // VEXT.32 d0,d0,d0,#0
uint32x2_t vext_u32(uint32x2_t a, uint32x2_t b, __constrange(0,1) int c); // VEXT.32 d0,d0,d0,#0
int64x1_t vext_s64(int64x1_t a, int64x1_t b, __constrange(0,0) int c); // VEXT.64 d0,d0,d0,#0
uint64x1_t vext_u64(uint64x1_t a, uint64x1_t b, __constrange(0,0) int c); // VEXT.64 d0,d0,d0,#0
int8x16_t vextq_s8(int8x16_t a, int8x16_t b, __constrange(0,15) int c); // VEXT.8 q0,q0,q0,#0
uint8x16_t vextq_u8(uint8x16_t a, uint8x16_t b, __constrange(0,15) int c); // VEXT.8 q0,q0,q0,#0
poly8x16_t vextq_p8(poly8x16_t a, poly8x16_t b, __constrange(0,15) int c); // VEXT.8 q0,q0,q0,#0
int16x8_t vextq_s16(int16x8_t a, int16x8_t b, __constrange(0,7) int c); // VEXT.16 q0,q0,q0,#0
uint16x8_t vextq_u16(uint16x8_t a, uint16x8_t b, __constrange(0,7) int c); // VEXT.16 q0,q0,q0,#0
poly16x8_t vextq_p16(poly16x8_t a, poly16x8_t b, __constrange(0,7) int c); // VEXT.16 q0,q0,q0,#0
int32x4_t vextq_s32(int32x4_t a, int32x4_t b, __constrange(0,3) int c); // VEXT.32 q0,q0,q0,#0
uint32x4_t vextq_u32(uint32x4_t a, uint32x4_t b, __constrange(0,3) int c); // VEXT.32 q0,q0,q0,#0
int64x2_t vextq_s64(int64x2_t a, int64x2_t b, __constrange(0,1) int c); // VEXT.64 q0,q0,q0,#0
uint64x2_t vextq_u64(uint64x2_t a, uint64x2_t b, __constrange(0,1) int c); // VEXT.64 q0,q0,q0,#0

```

G.32 Reverse vector elements (swap endianness)

VREVn.m reverses the order of the m-bit lanes within a set that is n bits wide.

```

int8x8_t      vrev64_s8(int8x8_t vec);      // VREV64.8 d0,d0
int16x4_t     vrev64_s16(int16x4_t vec);    // VREV64.16 d0,d0
int32x2_t     vrev64_s32(int32x2_t vec);    // VREV64.32 d0,d0
uint8x8_t     vrev64_u8(uint8x8_t vec);     // VREV64.8 d0,d0
uint16x4_t    vrev64_u16(uint16x4_t vec);   // VREV64.16 d0,d0
uint32x2_t    vrev64_u32(uint32x2_t vec);   // VREV64.32 d0,d0
poly8x8_t     vrev64_p8(poly8x8_t vec);     // VREV64.8 d0,d0
poly16x4_t    vrev64_p16(poly16x4_t vec);   // VREV64.16 d0,d0
float32x2_t   vrev64_f32(float32x2_t vec);  // VREV64.32 d0,d0
int8x16_t     vrev64q_s8(int8x16_t vec);    // VREV64.8 q0,q0
int16x8_t     vrev64q_s16(int16x8_t vec);   // VREV64.16 q0,q0
int32x4_t     vrev64q_s32(int32x4_t vec);   // VREV64.32 q0,q0
uint8x16_t    vrev64q_u8(uint8x16_t vec);   // VREV64.8 q0,q0
uint16x8_t    vrev64q_u16(uint16x8_t vec);  // VREV64.16 q0,q0
uint32x4_t    vrev64q_u32(uint32x4_t vec);  // VREV64.32 q0,q0
poly8x16_t    vrev64q_p8(poly8x16_t vec);   // VREV64.8 q0,q0
poly16x8_t    vrev64q_p16(poly16x8_t vec);  // VREV64.16 q0,q0
float32x4_t   vrev64q_f32(float32x4_t vec); // VREV64.32 q0,q0
int8x8_t      vrev32_s8(int8x8_t vec);      // VREV32.8 d0,d0
int16x4_t     vrev32_s16(int16x4_t vec);    // VREV32.16 d0,d0
uint8x8_t     vrev32_u8(uint8x8_t vec);     // VREV32.8 d0,d0
uint16x4_t    vrev32_u16(uint16x4_t vec);   // VREV32.16 d0,d0
poly8x8_t     vrev32_p8(poly8x8_t vec);     // VREV32.8 d0,d0
int8x16_t     vrev32q_s8(int8x16_t vec);    // VREV32.8 q0,q0
int16x8_t     vrev32q_s16(int16x8_t vec);   // VREV32.16 q0,q0
uint8x16_t    vrev32q_u8(uint8x16_t vec);   // VREV32.8 q0,q0
uint16x8_t    vrev32q_u16(uint16x8_t vec);  // VREV32.16 q0,q0
poly8x16_t    vrev32q_p8(poly8x16_t vec);   // VREV32.8 q0,q0
int8x8_t      vrev16_s8(int8x8_t vec);      // VREV16.8 d0,d0
uint8x8_t     vrev16_u8(uint8x8_t vec);     // VREV16.8 d0,d0
poly8x8_t     vrev16_p8(poly8x8_t vec);     // VREV16.8 d0,d0
int8x16_t     vrev16q_s8(int8x16_t vec);    // VREV16.8 q0,q0
uint8x16_t    vrev16q_u8(uint8x16_t vec);   // VREV16.8 q0,q0
poly8x16_t    vrev16q_p8(poly8x16_t vec);   // VREV16.8 q0,q0

```

G.33 Other single operand arithmetic

These intrinsics provide other single operand arithmetic.

G.33.1 Absolute: `vabs{q}_<type>`. `Vd[i] = |Va[i]|`

```
int8x8_t   vabs_s8(int8x8_t a);      // VABS.S8 d0,d0
int16x4_t  vabs_s16(int16x4_t a);   // VABS.S16 d0,d0
int32x2_t  vabs_s32(int32x2_t a);   // VABS.S32 d0,d0
float32x2_t vabs_f32(float32x2_t a); // VABS.F32 d0,d0
int8x16_t  vabsq_s8(int8x16_t a);   // VABS.S8 q0,q0
int16x8_t  vabsq_s16(int16x8_t a);  // VABS.S16 q0,q0
int32x4_t  vabsq_s32(int32x4_t a);  // VABS.S32 q0,q0
float32x4_t vabsq_f32(float32x4_t a); // VABS.F32 q0,q0
```

G.33.2 Saturating absolute: `vqabs{q}_<type>`. `Vd[i] = sat(|Va[i]|)`

```
int8x8_t   vqabs_s8(int8x8_t a);    // VQABS.S8 d0,d0
int16x4_t  vqabs_s16(int16x4_t a);  // VQABS.S16 d0,d0
int32x2_t  vqabs_s32(int32x2_t a);  // VQABS.S32 d0,d0
int8x16_t  vqabsq_s8(int8x16_t a);  // VQABS.S8 q0,q0
int16x8_t  vqabsq_s16(int16x8_t a); // VQABS.S16 q0,q0
int32x4_t  vqabsq_s32(int32x4_t a); // VQABS.S32 q0,q0
```

G.33.3 Negate: `vneg{q}_<type>`. `Vd[i] = - Va[i]`

```
int8x8_t   vneg_s8(int8x8_t a);     // VNEG.S8 d0,d0
int16x4_t  vneg_s16(int16x4_t a);   // VNEG.S16 d0,d0
int32x2_t  vneg_s32(int32x2_t a);   // VNEG.S32 d0,d0
float32x2_t vneg_f32(float32x2_t a); // VNEG.F32 d0,d0
int8x16_t  vnegq_s8(int8x16_t a);   // VNEG.S8 q0,q0
int16x8_t  vnegq_s16(int16x8_t a);  // VNEG.S16 q0,q0
int32x4_t  vnegq_s32(int32x4_t a);  // VNEG.S32 q0,q0
float32x4_t vnegq_f32(float32x4_t a); // VNEG.F32 q0,q0
```

G.33.4 Saturating Negate: `vqneg{q}_<type>`. `sat(Vd[i] = - Va[i])`

```
int8x8_t   vqneg_s8(int8x8_t a);    // VQNEG.S8 d0,d0
int16x4_t  vqneg_s16(int16x4_t a);  // VQNEG.S16 d0,d0
int32x2_t  vqneg_s32(int32x2_t a);  // VQNEG.S32 d0,d0
int8x16_t  vqnegq_s8(int8x16_t a);  // VQNEG.S8 q0,q0
int16x8_t  vqnegq_s16(int16x8_t a); // VQNEG.S16 q0,q0
int32x4_t  vqnegq_s32(int32x4_t a); // VQNEG.S32 q0,q0
```

G.33.5 Count leading sign bits

```
int8x8_t   vcls_s8(int8x8_t a);     // VCLS.S8 d0,d0
int16x4_t  vcls_s16(int16x4_t a);   // VCLS.S16 d0,d0
int32x2_t  vcls_s32(int32x2_t a);   // VCLS.S32 d0,d0
int8x16_t  vclsq_s8(int8x16_t a);   // VCLS.S8 q0,q0
int16x8_t  vclsq_s16(int16x8_t a);  // VCLS.S16 q0,q0
int32x4_t  vclsq_s32(int32x4_t a);  // VCLS.S32 q0,q0
```

G.33.6 Count leading zeros

```
int8x8_t   vclz_s8(int8x8_t a);     // VCLZ.I8 d0,d0
int16x4_t  vclz_s16(int16x4_t a);   // VCLZ.I16 d0,d0
int32x2_t  vclz_s32(int32x2_t a);   // VCLZ.I32 d0,d0
uint8x8_t  vclz_u8(uint8x8_t a);    // VCLZ.I8 d0,d0
uint16x4_t vclz_u16(uint16x4_t a);  // VCLZ.I16 d0,d0
```



```

uint32x2_t vclz_u32(uint32x2_t a); // VCLZ.I32 d0,d0
int8x16_t vclzq_s8(int8x16_t a); // VCLZ.I8 q0,q0
int16x8_t vclzq_s16(int16x8_t a); // VCLZ.I16 q0,q0
int32x4_t vclzq_s32(int32x4_t a); // VCLZ.I32 q0,q0
uint8x16_t vclzq_u8(uint8x16_t a); // VCLZ.I8 q0,q0
uint16x8_t vclzq_u16(uint16x8_t a); // VCLZ.I16 q0,q0
uint32x4_t vclzq_u32(uint32x4_t a); // VCLZ.I32 q0,q0

```

G.33.7 Count number of set bits

```

uint8x8_t vcnt_u8(uint8x8_t a); // VCNT.8 d0,d0
int8x8_t vcnt_s8(int8x8_t a); // VCNT.8 d0,d0
poly8x8_t vcnt_p8(poly8x8_t a); // VCNT.8 d0,d0
uint8x16_t vcntq_u8(uint8x16_t a); // VCNT.8 q0,q0
int8x16_t vcntq_s8(int8x16_t a); // VCNT.8 q0,q0
poly8x16_t vcntq_p8(poly8x16_t a); // VCNT.8 q0,q0

```

G.33.8 Reciprocal estimate

```

float32x2_t vrecpe_f32(float32x2_t a); // VRECPE.F32 d0,d0
uint32x2_t vrecpe_u32(uint32x2_t a); // VRECPE.U32 d0,d0
float32x4_t vrecpeq_f32(float32x4_t a); // VRECPE.F32 q0,q0
uint32x4_t vrecpeq_u32(uint32x4_t a); // VRECPE.U32 q0,q0

```

G.33.9 Reciprocal square root estimate

```

float32x2_t vrsqrte_f32(float32x2_t a); // VRSQRTE.F32 d0,d0
uint32x2_t vrsqrte_u32(uint32x2_t a); // VRSQRTE.U32 d0,d0
float32x4_t vrsqrteq_f32(float32x4_t a); // VRSQRTE.F32 q0,q0
uint32x4_t vrsqrteq_u32(uint32x4_t a); // VRSQRTE.U32 q0,q0

```

G.34 Logical operations

These intrinsics provide bitwise logical operations.

G.34.1 Bitwise not

```
int8x8_t   vmvn_s8(int8x8_t a);           // VMVN d0,d0
int16x4_t  vmvn_s16(int16x4_t a);        // VMVN d0,d0
int32x2_t  vmvn_s32(int32x2_t a);        // VMVN d0,d0
uint8x8_t  vmvn_u8(uint8x8_t a);         // VMVN d0,d0
uint16x4_t vmvn_u16(uint16x4_t a);       // VMVN d0,d0
uint32x2_t vmvn_u32(uint32x2_t a);       // VMVN d0,d0
poly8x8_t  vmvn_p8(poly8x8_t a);         // VMVN d0,d0
int8x16_t  vmvnq_s8(int8x16_t a);        // VMVN q0,q0
int16x8_t  vmvnq_s16(int16x8_t a);       // VMVN q0,q0
int32x4_t  vmvnq_s32(int32x4_t a);       // VMVN q0,q0
uint8x16_t vmvnq_u8(uint8x16_t a);       // VMVN q0,q0
uint16x8_t vmvnq_u16(uint16x8_t a);      // VMVN q0,q0
uint32x4_t vmvnq_u32(uint32x4_t a);      // VMVN q0,q0
poly8x16_t vmvnq_p8(poly8x16_t a);       // VMVN q0,q0
```

G.34.2 Bitwise and

```
int8x8_t   vand_s8(int8x8_t a, int8x8_t b); // VAND d0,d0,d0
int16x4_t  vand_s16(int16x4_t a, int16x4_t b); // VAND d0,d0,d0
int32x2_t  vand_s32(int32x2_t a, int32x2_t b); // VAND d0,d0,d0
int64x1_t  vand_s64(int64x1_t a, int64x1_t b); // VAND d0,d0,d0
uint8x8_t  vand_u8(uint8x8_t a, uint8x8_t b); // VAND d0,d0,d0
uint16x4_t vand_u16(uint16x4_t a, uint16x4_t b); // VAND d0,d0,d0
uint32x2_t vand_u32(uint32x2_t a, uint32x2_t b); // VAND d0,d0,d0
uint64x1_t vand_u64(uint64x1_t a, uint64x1_t b); // VAND d0,d0,d0
int8x16_t  vandq_s8(int8x16_t a, int8x16_t b); // VAND q0,q0,q0
int16x8_t  vandq_s16(int16x8_t a, int16x8_t b); // VAND q0,q0,q0
int32x4_t  vandq_s32(int32x4_t a, int32x4_t b); // VAND q0,q0,q0
int64x2_t  vandq_s64(int64x2_t a, int64x2_t b); // VAND q0,q0,q0
uint8x16_t vandq_u8(uint8x16_t a, uint8x16_t b); // VAND q0,q0,q0
uint16x8_t vandq_u16(uint16x8_t a, uint16x8_t b); // VAND q0,q0,q0
uint32x4_t vandq_u32(uint32x4_t a, uint32x4_t b); // VAND q0,q0,q0
uint64x2_t vandq_u64(uint64x2_t a, uint64x2_t b); // VAND q0,q0,q0
```

G.34.3 Bitwise or

```
int8x8_t   vorr_s8(int8x8_t a, int8x8_t b); // VORR d0,d0,d0
int16x4_t  vorr_s16(int16x4_t a, int16x4_t b); // VORR d0,d0,d0
int32x2_t  vorr_s32(int32x2_t a, int32x2_t b); // VORR d0,d0,d0
int64x1_t  vorr_s64(int64x1_t a, int64x1_t b); // VORR d0,d0,d0
uint8x8_t  vorr_u8(uint8x8_t a, uint8x8_t b); // VORR d0,d0,d0
uint16x4_t vorr_u16(uint16x4_t a, uint16x4_t b); // VORR d0,d0,d0
uint32x2_t vorr_u32(uint32x2_t a, uint32x2_t b); // VORR d0,d0,d0
uint64x1_t vorr_u64(uint64x1_t a, uint64x1_t b); // VORR d0,d0,d0
int8x16_t  vorrq_s8(int8x16_t a, int8x16_t b); // VORR q0,q0,q0
int16x8_t  vorrq_s16(int16x8_t a, int16x8_t b); // VORR q0,q0,q0
int32x4_t  vorrq_s32(int32x4_t a, int32x4_t b); // VORR q0,q0,q0
int64x2_t  vorrq_s64(int64x2_t a, int64x2_t b); // VORR q0,q0,q0
uint8x16_t vorrq_u8(uint8x16_t a, uint8x16_t b); // VORR q0,q0,q0
uint16x8_t vorrq_u16(uint16x8_t a, uint16x8_t b); // VORR q0,q0,q0
uint32x4_t vorrq_u32(uint32x4_t a, uint32x4_t b); // VORR q0,q0,q0
uint64x2_t vorrq_u64(uint64x2_t a, uint64x2_t b); // VORR q0,q0,q0
```

G.34.4 Bitwise exclusive or (EOR or XOR)

```

int8x8_t   veor_s8(int8x8_t a, int8x8_t b);           // VEOR d0,d0,d0
int16x4_t  veor_s16(int16x4_t a, int16x4_t b);       // VEOR d0,d0,d0
int32x2_t  veor_s32(int32x2_t a, int32x2_t b);       // VEOR d0,d0,d0
int64x1_t  veor_s64(int64x1_t a, int64x1_t b);       // VEOR d0,d0,d0
uint8x8_t  veor_u8(uint8x8_t a, uint8x8_t b);        // VEOR d0,d0,d0
uint16x4_t veor_u16(uint16x4_t a, uint16x4_t b);     // VEOR d0,d0,d0
uint32x2_t veor_u32(uint32x2_t a, uint32x2_t b);     // VEOR d0,d0,d0
uint64x1_t veor_u64(uint64x1_t a, uint64x1_t b);     // VEOR d0,d0,d0
int8x16_t  veorq_s8(int8x16_t a, int8x16_t b);      // VEOR q0,q0,q0
int16x8_t  veorq_s16(int16x8_t a, int16x8_t b);     // VEOR q0,q0,q0
int32x4_t  veorq_s32(int32x4_t a, int32x4_t b);     // VEOR q0,q0,q0
int64x2_t  veorq_s64(int64x2_t a, int64x2_t b);     // VEOR q0,q0,q0
uint8x16_t veorq_u8(uint8x16_t a, uint8x16_t b);    // VEOR q0,q0,q0
uint16x8_t veorq_u16(uint16x8_t a, uint16x8_t b);   // VEOR q0,q0,q0
uint32x4_t veorq_u32(uint32x4_t a, uint32x4_t b);   // VEOR q0,q0,q0
uint64x2_t veorq_u64(uint64x2_t a, uint64x2_t b);   // VEOR q0,q0,q0

```

G.34.5 Bit Clear

```

int8x8_t   vbic_s8(int8x8_t a, int8x8_t b);           // VBIC d0,d0,d0
int16x4_t  vbic_s16(int16x4_t a, int16x4_t b);       // VBIC d0,d0,d0
int32x2_t  vbic_s32(int32x2_t a, int32x2_t b);       // VBIC d0,d0,d0
int64x1_t  vbic_s64(int64x1_t a, int64x1_t b);       // VBIC d0,d0,d0
uint8x8_t  vbic_u8(uint8x8_t a, uint8x8_t b);        // VBIC d0,d0,d0
uint16x4_t vbic_u16(uint16x4_t a, uint16x4_t b);     // VBIC d0,d0,d0
uint32x2_t vbic_u32(uint32x2_t a, uint32x2_t b);     // VBIC d0,d0,d0
uint64x1_t vbic_u64(uint64x1_t a, uint64x1_t b);     // VBIC d0,d0,d0
int8x16_t  vbicq_s8(int8x16_t a, int8x16_t b);      // VBIC q0,q0,q0
int16x8_t  vbicq_s16(int16x8_t a, int16x8_t b);     // VBIC q0,q0,q0
int32x4_t  vbicq_s32(int32x4_t a, int32x4_t b);     // VBIC q0,q0,q0
int64x2_t  vbicq_s64(int64x2_t a, int64x2_t b);     // VBIC q0,q0,q0
uint8x16_t vbicq_u8(uint8x16_t a, uint8x16_t b);    // VBIC q0,q0,q0
uint16x8_t vbicq_u16(uint16x8_t a, uint16x8_t b);   // VBIC q0,q0,q0
uint32x4_t vbicq_u32(uint32x4_t a, uint32x4_t b);   // VBIC q0,q0,q0
uint64x2_t vbicq_u64(uint64x2_t a, uint64x2_t b);   // VBIC q0,q0,q0

```

G.34.6 Bitwise OR complement

```

int8x8_t   vorn_s8(int8x8_t a, int8x8_t b);           // VORN d0,d0,d0
int16x4_t  vorn_s16(int16x4_t a, int16x4_t b);       // VORN d0,d0,d0
int32x2_t  vorn_s32(int32x2_t a, int32x2_t b);       // VORN d0,d0,d0
int64x1_t  vorn_s64(int64x1_t a, int64x1_t b);       // VORN d0,d0,d0
uint8x8_t  vorn_u8(uint8x8_t a, uint8x8_t b);        // VORN d0,d0,d0
uint16x4_t vorn_u16(uint16x4_t a, uint16x4_t b);     // VORN d0,d0,d0
uint32x2_t vorn_u32(uint32x2_t a, uint32x2_t b);     // VORN d0,d0,d0
uint64x1_t vorn_u64(uint64x1_t a, uint64x1_t b);     // VORN d0,d0,d0
int8x16_t  vornq_s8(int8x16_t a, int8x16_t b);      // VORN q0,q0,q0
int16x8_t  vornq_s16(int16x8_t a, int16x8_t b);     // VORN q0,q0,q0
int32x4_t  vornq_s32(int32x4_t a, int32x4_t b);     // VORN q0,q0,q0
int64x2_t  vornq_s64(int64x2_t a, int64x2_t b);     // VORN q0,q0,q0
uint8x16_t vornq_u8(uint8x16_t a, uint8x16_t b);    // VORN q0,q0,q0
uint16x8_t vornq_u16(uint16x8_t a, uint16x8_t b);   // VORN q0,q0,q0
uint32x4_t vornq_u32(uint32x4_t a, uint32x4_t b);   // VORN q0,q0,q0
uint64x2_t vornq_u64(uint64x2_t a, uint64x2_t b);   // VORN q0,q0,q0

```

G.34.7 Bitwise Select

————— **Note** —————

This intrinsic can compile to any of VBSL/VBIF/VBIT depending on register allocation.

```

int8x8_t    vbsl_s8(uint8x8_t a, int8x8_t b, int8x8_t c);           // VBSL d0,d0,d0
int16x4_t   vbsl_s16(uint16x4_t a, int16x4_t b, int16x4_t c);      // VBSL d0,d0,d0
int32x2_t   vbsl_s32(uint32x2_t a, int32x2_t b, int32x2_t c);      // VBSL d0,d0,d0
int64x1_t   vbsl_s64(uint64x1_t a, int64x1_t b, int64x1_t c);      // VBSL d0,d0,d0
uint8x8_t   vbsl_u8(uint8x8_t a, uint8x8_t b, uint8x8_t c);        // VBSL d0,d0,d0
uint16x4_t  vbsl_u16(uint16x4_t a, uint16x4_t b, uint16x4_t c);     // VBSL d0,d0,d0
uint32x2_t  vbsl_u32(uint32x2_t a, uint32x2_t b, uint32x2_t c);     // VBSL d0,d0,d0
uint64x1_t  vbsl_u64(uint64x1_t a, uint64x1_t b, uint64x1_t c);     // VBSL d0,d0,d0
float32x2_t vbsl_f32(uint32x2_t a, float32x2_t b, float32x2_t c);   // VBSL d0,d0,d0
poly8x8_t   vbsl_p8(uint8x8_t a, poly8x8_t b, poly8x8_t c);        // VBSL d0,d0,d0
poly16x4_t  vbsl_p16(uint16x4_t a, poly16x4_t b, poly16x4_t c);     // VBSL d0,d0,d0
int8x16_t   vbslq_s8(uint8x16_t a, int8x16_t b, int8x16_t c);      // VBSL q0,q0,q0
int16x8_t   vbslq_s16(uint16x8_t a, int16x8_t b, int16x8_t c);     // VBSL q0,q0,q0
int32x4_t   vbslq_s32(uint32x4_t a, int32x4_t b, int32x4_t c);     // VBSL q0,q0,q0
int64x2_t   vbslq_s64(uint64x2_t a, int64x2_t b, int64x2_t c);     // VBSL q0,q0,q0
uint8x16_t  vbslq_u8(uint8x16_t a, uint8x16_t b, uint8x16_t c);     // VBSL q0,q0,q0
uint16x8_t  vbslq_u16(uint16x8_t a, uint16x8_t b, uint16x8_t c);    // VBSL q0,q0,q0
uint32x4_t  vbslq_u32(uint32x4_t a, uint32x4_t b, uint32x4_t c);    // VBSL q0,q0,q0
uint64x2_t  vbslq_u64(uint64x2_t a, uint64x2_t b, uint64x2_t c);    // VBSL q0,q0,q0
float32x4_t vbslq_f32(uint32x4_t a, float32x4_t b, float32x4_t c);  // VBSL q0,q0,q0
poly8x16_t  vbslq_p8(uint8x16_t a, poly8x16_t b, poly8x16_t c);    // VBSL q0,q0,q0
poly16x8_t  vbslq_p16(uint16x8_t a, poly16x8_t b, poly16x8_t c);    // VBSL q0,q0,q0

```

G.35 Transposition operations

These intrinsics provide transposition operations.

G.35.1 Transpose elements

```
int8x8x2_t   vtrn_s8(int8x8_t a, int8x8_t b);           // VTRN.8 d0,d0
int16x4x2_t  vtrn_s16(int16x4_t a, int16x4_t b);        // VTRN.16 d0,d0
int32x2x2_t  vtrn_s32(int32x2_t a, int32x2_t b);        // VTRN.32 d0,d0
uint8x8x2_t  vtrn_u8(uint8x8_t a, uint8x8_t b);         // VTRN.8 d0,d0
uint16x4x2_t vtrn_u16(uint16x4_t a, uint16x4_t b);      // VTRN.16 d0,d0
uint32x2x2_t vtrn_u32(uint32x2_t a, uint32x2_t b);      // VTRN.32 d0,d0
float32x2x2_t vtrn_f32(float32x2_t a, float32x2_t b);   // VTRN.32 d0,d0
poly8x8x2_t  vtrn_p8(poly8x8_t a, poly8x8_t b);         // VTRN.8 d0,d0
poly16x4x2_t vtrn_p16(poly16x4_t a, poly16x4_t b);      // VTRN.16 d0,d0
int8x16x2_t  vtrnq_s8(int8x16_t a, int8x16_t b);       // VTRN.8 q0,q0
int16x8x2_t  vtrnq_s16(int16x8_t a, int16x8_t b);      // VTRN.16 q0,q0
int32x4x2_t  vtrnq_s32(int32x4_t a, int32x4_t b);      // VTRN.32 q0,q0
uint8x16x2_t vtrnq_u8(uint8x16_t a, uint8x16_t b);     // VTRN.8 q0,q0
uint16x8x2_t vtrnq_u16(uint16x8_t a, uint16x8_t b);    // VTRN.16 q0,q0
uint32x4x2_t vtrnq_u32(uint32x4_t a, uint32x4_t b);    // VTRN.32 q0,q0
float32x4x2_t vtrnq_f32(float32x4_t a, float32x4_t b); // VTRN.32 q0,q0
poly8x16x2_t vtrnq_p8(poly8x16_t a, poly8x16_t b);     // VTRN.8 q0,q0
poly16x8x2_t vtrnq_p16(poly16x8_t a, poly16x8_t b);    // VTRN.16 q0,q0
```

G.35.2 Interleave elements

```
int8x8x2_t   vzip_s8(int8x8_t a, int8x8_t b);           // VZIP.8 d0,d0
int16x4x2_t  vzip_s16(int16x4_t a, int16x4_t b);        // VZIP.16 d0,d0
uint8x8x2_t  vzip_u8(uint8x8_t a, uint8x8_t b);         // VZIP.8 d0,d0
uint16x4x2_t vzip_u16(uint16x4_t a, uint16x4_t b);      // VZIP.16 d0,d0
float32x2x2_t vzip_f32(float32x2_t a, float32x2_t b);   // VZIP.32 d0,d0
poly8x8x2_t  vzip_p8(poly8x8_t a, poly8x8_t b);         // VZIP.8 d0,d0
poly16x4x2_t vzip_p16(poly16x4_t a, poly16x4_t b);      // VZIP.16 d0,d0
int8x16x2_t  vzipq_s8(int8x16_t a, int8x16_t b);       // VZIP.8 q0,q0
int16x8x2_t  vzipq_s16(int16x8_t a, int16x8_t b);      // VZIP.16 q0,q0
int32x4x2_t  vzipq_s32(int32x4_t a, int32x4_t b);      // VZIP.32 q0,q0
uint8x16x2_t vzipq_u8(uint8x16_t a, uint8x16_t b);     // VZIP.8 q0,q0
uint16x8x2_t vzipq_u16(uint16x8_t a, uint16x8_t b);    // VZIP.16 q0,q0
uint32x4x2_t vzipq_u32(uint32x4_t a, uint32x4_t b);    // VZIP.32 q0,q0
float32x4x2_t vzipq_f32(float32x4_t a, float32x4_t b); // VZIP.32 q0,q0
poly8x16x2_t vzipq_p8(poly8x16_t a, poly8x16_t b);     // VZIP.8 q0,q0
poly16x8x2_t vzipq_p16(poly16x8_t a, poly16x8_t b);    // VZIP.16 q0,q0
```

G.35.3 De-Interleave elements

```
int8x8x2_t   vuzp_s8(int8x8_t a, int8x8_t b);           // VUZP.8 d0,d0
int16x4x2_t  vuzp_s16(int16x4_t a, int16x4_t b);        // VUZP.16 d0,d0
int32x2x2_t  vuzp_s32(int32x2_t a, int32x2_t b);        // VUZP.32 d0,d0
uint8x8x2_t  vuzp_u8(uint8x8_t a, uint8x8_t b);         // VUZP.8 d0,d0
uint16x4x2_t vuzp_u16(uint16x4_t a, uint16x4_t b);      // VUZP.16 d0,d0
uint32x2x2_t vuzp_u32(uint32x2_t a, uint32x2_t b);      // VUZP.32 d0,d0
float32x2x2_t vuzp_f32(float32x2_t a, float32x2_t b);   // VUZP.32 d0,d0
poly8x8x2_t  vuzp_p8(poly8x8_t a, poly8x8_t b);         // VUZP.8 d0,d0
poly16x4x2_t vuzp_p16(poly16x4_t a, poly16x4_t b);      // VUZP.16 d0,d0
int8x16x2_t  vuzpq_s8(int8x16_t a, int8x16_t b);       // VUZP.8 q0,q0
int16x8x2_t  vuzpq_s16(int16x8_t a, int16x8_t b);      // VUZP.16 q0,q0
int32x4x2_t  vuzpq_s32(int32x4_t a, int32x4_t b);      // VUZP.32 q0,q0
uint8x16x2_t vuzpq_u8(uint8x16_t a, uint8x16_t b);     // VUZP.8 q0,q0
uint16x8x2_t vuzpq_u16(uint16x8_t a, uint16x8_t b);    // VUZP.16 q0,q0
uint32x4x2_t vuzpq_u32(uint32x4_t a, uint32x4_t b);    // VUZP.32 q0,q0
```

```
float32x4x2_t vuzpq_f32(float32x4_t a, float32x4_t b); // VUZP.32 q0,q0  
poly8x16x2_t vuzpq_p8(poly8x16_t a, poly8x16_t b); // VUZP.8 q0,q0  
poly16x8x2_t vuzpq_p16(poly16x8_t a, poly16x8_t b); // VUZP.16 q0,q0
```

G.36 Vector reinterpret cast operations

In some situations, you might want to treat a vector as having a different type, without changing its value. A set of intrinsics is provided to perform this type of conversion.

G.36.1 Syntax

```
vreinterpret{q}_dsttype_srctype
```

Where:

q Specifies that the conversion operates on 128-bit vectors. If it is not present, the conversion operates on 64-bit vectors.

dsttype Represents the type to convert to.

srctype Represents the type being converted.

G.36.2 Example

The following intrinsic reinterprets a vector of four signed 16-bit integers as a vector of four unsigned integers:

```
uint16x4_t vreinterpret_u16_s16(int16x4_t a);
```

The following intrinsic reinterprets a vector of four 32-bit floating point values integers as a vector of four signed integers.

```
int8x16_t vreinterpretq_s8_f32(float32x4_t a);
```

These conversions do not change the bit pattern represented by the vector.

Appendix H

Revisions for the Compiler Reference

This appendix describes the technical changes between released issues of this book.

Table H-1 Differences between issue G and issue H

Change	Topics affected
Added usage subtopics.	<ul style="list-style-type: none"> • #pragma arm on page 5-87. • #pragma thumb on page 5-112.
Documented Cortex-M0+ support.	• --cpu=name on page 3-49.
Modified the description of the generation of RTTI data with <code>--no_rtti_data</code> .	• --rtti_data , --no_rtti_data on page 3-186.
Clarified the different treatments of functions with side effects compared to those without them, by intrinsics that set scheduling barriers.	<ul style="list-style-type: none"> • __nop intrinsic on page 5-138. • __schedule_barrier intrinsic on page 5-152.
Added a note about a possible issue caused by overlapping diagnostic message number ranges.	<ul style="list-style-type: none"> • --diag_error=tag[,tag,...] on page 3-70. • --diag_remark=tag[,tag,...] on page 3-71. • --diag_suppress=tag[,tag,...] on page 3-73. • --diag_warning=tag[,tag,...] on page 3-75.

Table H-2 Differences between issue F and issue G

Change	Topics affected
Clarified the difference between <code>__packed</code> and <code>#pragma pack</code> for address-taken fields.	<ul style="list-style-type: none"> • __packed on page 5-17. • __attribute((packed)) type attribute on page 5-68. • #pragma pack(n) on page 5-107.
Mentioned that the compiler recognizes the pragmas <code>STDC CX_LIMITED_RANGE</code> , <code>STDC FENV_ACCESS</code> , and <code>STDC FP_CONTRACT</code> , but does not support them.	Preprocessing directives on page D-15.
Where appropriate, rather than 16-bit Thumb or 32-bit Thumb, referred instead to Thumb, or Thumb-2 technology.	Various topics.
Noted that <code>armcc -E</code> disables implicit inclusion.	-E on page 3-82.
Added two entries for VFPv4, under <code>__TARGET_FPU_xx</code> .	<ul style="list-style-type: none"> • Predefined macros on page 5-183. • Table 5-21 on page 5-183.
Added restrictions on C55x intrinsic support.	C55x intrinsics on page 5-171.
Mentioned that GNU mode also affects C/C++ standards compliance.	--gnu on page 3-107.
Modified the part of the usage section discouraging expressions with side effects, and clarified the text.	__promise intrinsic on page 5-144.
Removed a reference to not accessing a physical register directly.	--use_frame_pointer on page 3-209.
Added Cortex-A7.	<ul style="list-style-type: none"> • Supported ARM architectures on page 3-49. • ARMv6 SIMD intrinsics, compatible processors and architectures on page A-9. • Predefined macros on page 5-183.
Modified the usage section for the <code>__weak</code> keyword for cases of multiple weak definitions.	__weak on page 5-27.
Mentioned that NaNs used with the <code>--fpmode=std</code> or <code>--fpmode=fast</code> option can produce undefined behavior.	--fpmode=model on page 3-97.
Added a usage section for function attributes.	Function attributes on page 5-39.
Reduced use of 32-bit Thumb in favor of Thumb or Thumb-2 technology.	Various topics.

Table H-3 Differences between issue E and issue F

Change	Topics affected
Added a note stating that the <code>--device</code> option is deprecated.	<ul style="list-style-type: none"> • --device=list on page 3-68. • --device=name on page 3-69.
Added a note about the maximum version of <code>gcc</code> supported by <code>armcc</code> .	--gnu_version=version on page 3-110.
Mentioned that downgradeable errors are also suppressed with <code>-J</code> .	-Jdir[,dir,...] on page 3-125.

Table H-3 Differences between issue E and issue F (continued)

Change	Topics affected
Modified the description for the <code>--licretry</code> option.	--licretry on page 3-131.
Added the <code>--protect_stack</code> option.	--protect_stack , --no_protect_stack on page 3-176.
Modified the description for the <code>--version_number</code> option.	--version_number on page 3-214.
Modified the description for the <code>--vsn</code> option.	--vsn on page 3-219.
Changed the format description for <code>__ARMCC_VERSION</code> .	Predefined macros on page 5-183.
Where appropriate: <ul style="list-style-type: none"> changed Thumb-1 to 16-bit Thumb changed Thumb-2 to 32-bit Thumb changed Thumb-2EE to ThumbEE. 	Various topics.
Changed the <code>ARMCCnn_CCOPT</code> and <code>ARMCCnnINC</code> environment variables to <code>ARMCCn_CCOPT</code> and <code>ARMCCnINC</code> .	Various topics.

Table H-4 Differences between issue D and issue E

Change	Topics affected
Improved usage description of <code>--reassociate_saturation</code> .	--reassociate_saturation , --no_reassociate_saturation on page 3-177.
Added the encoding details of opcodes and registers.	__cdp_intrinsic on page 5-119.
Added the options <code>--allow_fpreg_for_nonfpdata</code> , and <code>--no_allow_fpreg_for_nonfpdata</code> .	--allow_fpreg_for_nonfpdata , --no_allow_fpreg_for_nonfpdata on page 3-7.
Added the options <code>--conditionalize</code> , and <code>--no_conditionalize</code> .	--conditionalize , --no_conditionalize on page 3-38.
Added SC000 to the table of <code>--cpu</code> options.	--cpu=name on page 3-49.
Added SC300 and SC000 to the table of <code>--compatible</code> options.	--compatible=name on page 3-36.
Changed <code>--depend=filename</code> to say that for multiple files, the generated dependency file contains dependency lines from all the source files.	--depend=filename on page 3-61.
Added a caution that volatile is ignored if used with the <code>__global_reg</code> storage class specifier.	__global_reg on page 5-11.

Table H-5 Differences between issue C and issue D

Change	Topics affected
Added a summary table of intrinsics and their page numbers.	Summary of NEON intrinsics on page G-5.
Added syntaxes and tables that provide a condensed representation of the intrinsics. Fixed minor errors in the intrinsic prototypes.	Intrinsics on page G-10.
Removed the <code>--profile</code> option.	Chapter 3 Compiler Command-line Options .
Added list of built-in GNU atomic memory access functions.	GNU built-in functions on page 5-180.
Added Cortex-A15 and Cortex-R7 to the cpu list.	<ul style="list-style-type: none"> • Supported ARM architectures on page 3-49. • ARMv6 SIMD intrinsics, compatible processors and architectures on page A-9. • Predefined macros on page 5-183.
Changed option from <code>--implicit_using_std</code> to <code>--using_std</code> .	Table 5-21 on page 5-183.
Added v7E-M to table of Thumb architecture versions in relation to ARM architecture versions.	Table 5-22 on page 5-188.
Added a note that some registers are not available on some architectures.	Named register variables on page 5-176.
Added the <code>--echo</code> option.	<code>--echo</code> on page 3-83.
Added the <code>--use_frame_pointer</code> option.	<code>--use_frame_pointer</code> on page 3-209.
Added the <code>--depend_single_line</code> and <code>--no_depend_single_line</code> options.	<ul style="list-style-type: none"> • <code>--depend=filename</code> on page 3-61. • <code>--depend_single_line</code>, <code>--no_depend_single_line</code> on page 3-65. • <code>-M</code> on page 3-144. • <code>--md</code> on page 3-145.
Changed ARMCC41* environment variables to ARMCCnn*.	Various topics.
Changed ARM Compiler v4.1 to ARM Compiler 4.1 and later.	<ul style="list-style-type: none"> • Chapter 3 Compiler Command-line Options. • Predefined macros on page 5-183. • Structures, unions, enumerations, and bitfields on page 6-9. • Tentative arrays on page 6-16.
Added the <code>--library_interface=none</code> option.	<code>--library_interface=lib</code> on page 3-128.
Added the <code>--preprocess_assembly</code> option.	<code>--preprocess_assembly</code> on page 3-173.
Added the <code>-Warmcc</code> , <code>--gcc_fallback</code> option.	<code>-Warmcc</code> , <code>--gcc_fallback</code> on page 3-222.
Modified description of <code>--remove_unneeded_entities</code> option.	<code>--remove_unneeded_entities</code> , <code>--no_remove_unneeded_entities</code> on page 3-182.
Changed <code>--apcs</code> options to use variable list.	<code>--apcs=qualifer...qualifier</code> on page 3-11.

Table H-5 Differences between issue C and issue D (continued)

Change	Topics affected
Added note that the option is not required if you are using the ARM Compiler toolchain with DS-5.	<ul style="list-style-type: none"> • --reinitialize_workdir on page 3-179. • --workdir=directory on page 3-227. • --project=filename, --no_project on page 3-175.
Added link to command line options and search paths.	<ul style="list-style-type: none"> • -Jdir[,dir,...] on page 3-125. • -Idir[,dir,...] on page 3-114.
Added <code>./include</code> as a search path.	--implicit_include_searches, --no_implicit_include_searches on page 3-117.
Added <code>const unsigned int</code> for the variable declarations in the <code>__cdp</code> example. Also changed the value of <code>ops</code> from <code>0xAB</code> to <code>0xA3</code> because the second opcode must be 3 bits.	__cdp intrinsic on page 5-119.
Changed the note to say that the <code>__irq</code> function compiles to ARM or Thumb code when compiling for a processor that supports ARM and 32-bit Thumb. Added links to <code>--arm, #pragma arm, and ARM, Thumb, and ThumbEE instruction sets</code> .	__irq on page 5-16.
Mentioned that PC is set to LR-4 only in architectures other than ARMv6-M and ARMv7-M. And added note that for ARMv6-M and ARMv7-M, <code>__irq</code> does not affect the compiled output.	__irq on page 5-16.
Added note to deprecate <code>--ltcg</code> . Also added links to <code>ltcg</code> topics in <i>Using the Compiler</i> .	--ltcg on page 3-143.
Changed <code>FPv4_SP</code> to <code>FPv4-SP</code> .	--fpu=name on page 3-100.
Added ARM Glossary to other information.	Chapter 1 Conventions and Feedback .
Removed <code>#pragma GCC visibility from --visibility_inlines_hidden</code> .	--visibility_inlines_hidden on page 3-217.
Added detail about <code>mask</code> and <code>flags</code> bit. Also added note and link to <code><fenv.h></code> topic in <i>Using the Compiler</i> . Changed "preferable" to "ARM recommends".	VFP status intrinsic on page 5-173.
Mentioned class, struct, union, and enum types in the Usage section. Also added that you can apply this attribute to functions and variables.	<ul style="list-style-type: none"> • __attribute__((visibility("visibility_type"))) variable attribute on page 5-81. • __attribute__((visibility("visibility_type"))) function attribute on page 5-61.
Changed <code>--vfp</code> to <code>--fpu</code> .	--fpu=name on page 3-100.
Corrected description of the <code>--depend</code> option when specifying multiple source files.	--depend=filename on page 3-61.

Table H-6 Differences between issue B and issue C

Change	Topics affected
Updated the Modes supported column, for example changed Standard C90 to C90. Added GNU C++ to Compound literals. Added C90, C99, C++ to Variadic macros. Changed the origin of <code>__alignof__</code> to GCC-specific. Removed GNU C++ from void pointer arithmetic.	Supported GNU extensions on page C-1.
Removed the mention of the modes (C90 and C++) from the list of the Standard C99 features.	GNU extensions to the C and C++ languages on page 4-48.
Removed <code>asm</code> keyword from the list of features that are not part of the ISO standard. This is because the <code>asm</code> keyword is part of Standard C++. The <code>asm</code> keyword is mentioned separately.	GNU extensions to the C and C++ languages on page 4-48.
Renamed the column Extension origin to Origin. Mentioned GCC-specific in the Origin column for the entries on <code>__attribute__*</code> .	Supported GNU extensions on page C-1.

Table H-7 Differences between issue A and issue B

Change	Topics affected
Compiler faults use of <code>at</code> attribute when used on declarations with incomplete types.	__attribute__((at(address))) variable attribute on page 5-72.
Input parameter descriptions. User guidance that this intrinsic is for expert use only.	__cdp intrinsic on page 5-119.
Return value saturated to unsigned range $0 \leq x \leq 2^{sat} - 1$.	__usat intrinsic on page 5-164.
Introductory and usage descriptions.	__promise intrinsic on page 5-144.
<code>--ignore_missing_headers</code> only takes effect when dependency generation options are specified.	--ignore_missing_headers on page 3-115.
Descriptive clarification for <code>rvct30</code> , <code>rvct30_c90</code> , <code>rvct31</code> , <code>rvct31_c90</code> , <code>rvct40</code> , <code>rvct40_c90</code> .	--library_interface=lib on page 3-128.
If using <code>--show_cmdline</code> with ARM Linux translation options, you must use <code>-warmcc</code> .	--show_cmdline on page 3-189.
Cases where <code>--show_cmdline</code> can be useful.	--show_cmdline on page 3-189.
Clarification that <code>--default_definition_visibility=visibility</code> controls the default ELF symbol visibility of <code>extern</code> variable and function definitions.	--default_definition_visibility=visibility on page 3-58.
<code>__declspec(dllimport)</code> imports a symbol through the dynamic symbol table when <i>linking against</i> DLL libraries. (Textual clarification only.)	__declspec(dllimport) on page 5-33.
New topic.	--narrow_volatile_bitfields on page 3-152.
Added APSR, PSR, DSP, MVFR1, MVFR0, FPINST, FPINST2.	Named register variables on page 5-176.
Additional GNU built-in functions.	Nonstandard functions on page 5-180.

Table H-7 Differences between issue A and issue B (continued)

Change	Topics affected
Clarification to restrictions on use of <code>__packed</code> when casting.	__packed on page 5-17.
Added ARM v7E-M architecture, example processor Cortex-M4.	--cpu=name on page 3-49.
Added <code>__TARGET_FEATURE_NEON</code> .	Predefined macros on page 5-183.
New function attribute that is a GNU compiler extension supported by the ARM compiler.	__attribute__((format_arg(string-index))) function attribute on page 5-48.
Default option depends on optimization level.	--data_reorder , --no_data_reorder on page 3-55.
Removed "The keyword <code>__align</code> comes immediately before the variable name", because both of the following are now compilable: <code>__align(n) static int x;</code> <code>static __align(n) int x;</code>	__align on page 5-6.
GNU extensions to the C and C++ languages.	<ul style="list-style-type: none"> • GNU extensions to the C and C++ languages on page 4-48. • Appendix C Summary Table of GNU Language Extensions.
Restrictions clarification.	--fpu=name on page 3-100.
Default option is independent of the optimization level.	--multifile , --no_multifile on page 3-150.
Optimization level is independent of multifile compilation.	-Onum on page 3-156.
Options are not necessarily restricted to vectorization usage.	--reassociate_saturation , --no_reassociate_saturation on page 3-177.
Removed from document. Available as knowledgebase articles. See Appendix C Summary Table of GNU Language Extensions .	<code>asm</code> keyword, case ranges, cast of a union, character escape sequences, compound literals, conditionals, designated inits, extended lvalues, initializers, inline functions, labels as values, pointer arithmetic, statement expressions, unnamed fields, zero-length arrays.
Textual clarification.	__wfi intrinsic on page 5-166.
Textual clarification.	__yield intrinsic on page 5-167.
Changed the value of the modulo result for four intrinsic functions from modulo ⁶⁴ to modulo 2 ⁶⁴ .	<ul style="list-style-type: none"> • __smlald intrinsic on page A-29. • __smlaldx intrinsic on page A-30. • __smlsld intrinsic on page A-33. • __smlsldx intrinsic on page A-34.
Changed addition to subtraction for both of the val options in the <code>__ssub16</code> intrinsic.	__ssub16 intrinsic on page A-41.