ARM® Compiler v5.06 for µVision®

Version 5

armcc User Guide

Confidential - Draft - Beta

ARM DUI0375G_02

ARM® Compiler v5.06 for µVision®

armcc User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
A	May 2007	Non-Confidential	Release for RVCT v3.1 Release for µVision
B	December 2008	Non-Confidential	Release for RVCT v4.0 Release for µVision
C	June 2011	Non-Confidential	Release for ARM Compiler v4.1 for µVision
D	July 2012	Non-Confidential	Release for ARM Compiler v5.02 for µVision
E	30 May 2014	Non-Confidential	Release for ARM Compiler v5.04 for µVision
F	12 December 2014	Non-Confidential	Release for ARM Compiler v5.05 for µVision
G-02	15 August 2015	Confidential - Draft	Release for ARM Compiler v5.06 for µVision

Additional Notices

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 Confidential. This document may only be used and distributed 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 for a Beta product, that is a product under development.

Web Address

http://www.arm.com

Contents

ARM® Compiler v5.06 for µVision® armcc User Guide

Preface

About this book 24

Chapter 1 Overview of the Compiler

1.1 The compiler 1-28
1.2 Source language modes of the compiler 1-29
1.3 Language extensions 1-31
1.4 Language compliance 1-32
1.5 The C and C++ libraries 1-33

Chapter 2 Getting Started with the Compiler

2.1 Compiler command-line syntax 2-35
2.2 Compiler command-line options listed by group 2-36
2.3 Default compiler behavior 2-42
2.4 Order of compiler command-line options 2-43
2.5 Using stdin to input source code to the compiler 2-44
2.6 Directing output to stdout 2-46
2.7 Filename suffixes recognized by the compiler 2-47
2.8 Compiler output files 2-49
2.9 Factors influencing how the compiler searches for header files 2-50
2.10 Compiler command-line options and search paths 2-51
2.11 Compiler search rules and the current place 2-52

2.12 The ARMCC5INC environment variable 2-53
2.13 Code compatibility between separately compiled and assembled modules 2-54
2.14 Linker feedback during compilation 2-55
2.15 Unused function code 2-56
2.16 Minimizing code size by eliminating unused functions during compilation 2-57
2.17 Compilation build time 2-58

Chapter 3 Compiler Features

3.1 Compiler intrinsics 3-64
3.2 Performance benefits of compiler intrinsics 3-65
3.3 ARM assembler instruction intrinsics 3-66
3.4 Generic intrinsics 3-67
3.5 Compiler intrinsics for controlling IRQ and FIQ interrupts 3-68
3.6 Compiler intrinsics for inserting optimization barriers 3-69
3.7 Compiler intrinsics for inserting native instructions 3-70
3.8 Compiler intrinsics for Digital Signal Processing (DSP) 3-71
3.9 Compiler support for European Telecommunications Standards Institute (ETSI) basic operations 3-72
3.10 Overflow and carry status flags for C and C++ code 3-74
3.11 Texas Instruments (TI) C55x intrinsics for optimizing C code 3-75
3.12 Compiler support for accessing registers using named register variables 3-76
3.13 Pragmas recognized by the compiler 3-79
3.14 Compiler and processor support for bit-banding 3-81
3.15 Compiler type attribute, __attribute__((bitband)) 3-82
3.16 --bitband compiler command-line option 3-83
3.17 How the compiler handles bit-band objects placed outside bit-band regions 3-84
3.18 Compiler support for thread-local storage 3-85
3.19 Compiler support for literal pools 3-86
3.20 Compiler eight-byte alignment features 3-87
3.21 Precompiled Header (PCH) files 3-88
3.22 Automatic Precompiled Header (PCH) file processing 3-90
3.23 Precompiled Header (PCH) file processing and the header stop point 3-91
3.24 Precompiled Header (PCH) file creation requirements 3-93
3.25 Compilation with multiple Precompiled Header (PCH) files 3-95
3.26 Obsolete Precompiled Header (PCH) files 3-96
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file 3-97
3.28 Selectively applying Precompiled Header (PCH) file processing 3-98
3.29 Suppressing Precompiled Header (PCH) file processing 3-99
3.30 Message output during Precompiled Header (PCH) processing 3-100
3.31 Performance issues with Precompiled Header (PCH) files 3-101
3.32 Default compiler options that are affected by optimization level 3-102

Chapter 4 Compiler Coding Practices

4.1 The compiler as an optimizing compiler 4-106
4.2 Compiler optimization for code size versus speed 4-107
4.3 Compiler optimization levels and the debug view 4-108
4.4 Selecting the target processor at compile time 4-111
4.5 Enabling FPU for bare-metal 4-112
4.6 Optimization of loop termination in C code 4-113

4.7 Loop unrolling in C code 4-115
4.8 Compiler optimization and the volatile keyword 4-117
4.9 Code metrics 4-119
4.10 Code metrics for measurement of code size and data size 4-120
4.11 Stack use in C and C++ 4-121
4.12 Benefits of reducing debug information in objects and libraries 4-123
4.13 Methods of reducing debug information in objects and libraries 4-124
4.14 Guarding against multiple inclusion of header files 4-125
4.15 Methods of minimizing function parameter passing overhead 4-126
4.16 Returning structures from functions through registers 4-127
4.17 Functions that return the same result when called with the same arguments 4-128
4.18 Comparison of pure and impure functions 4-129
4.19 Recommendation of postfix syntax when qualifying functions with ARM function modifiers 4-130
4.20 Inline functions 4-131
4.21 Compiler decisions on function inlining 4-132
4.22 Automatic function inlining and static functions 4-133
4.23 Inline functions and removal of unused out-of-line functions at link time 4-134
4.24 Automatic function inlining and multifile compilation 4-135
4.25 Restriction on overriding compiler decisions about function inlining 4-136
4.26 Compiler modes and inline functions 4-137
4.27 Inline functions in C++ and C90 mode 4-138
4.28 Inline functions in C99 mode 4-139
4.29 Inline functions and debugging 4-141
4.30 Types of data alignment 4-142
4.31 Advantages of natural data alignment 4-143
4.32 Compiler storage of data objects by natural byte alignment 4-144
4.33 Relevance of natural data alignment at compile time 4-145
4.34 Unaligned data access in C and C++ code 4-146
4.35 The __packed qualifier and unaligned data access in C and C++ code 4-147
4.36 Unaligned fields in structures 4-148
4.37 Performance penalty associated with marking whole structures as packed 4-149
4.38 Unaligned pointers in C and C++ code 4-150
4.39 Unaligned Load Register (LDR) instructions generated by the compiler 4-151
4.40 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 4-152
4.41 Compiler support for floating-point arithmetic 4-154
4.42 Default selection of hardware or software floating-point support 4-156
4.43 Example of hardware and software support differences for floating-point arithmetic 4-157
4.44 Vector Floating-Point (VFP) architectures 4-159
4.45 Limitations on hardware handling of floating-point arithmetic 4-160
4.46 Implementation of Vector Floating-Point (VFP) support code 4-161
4.47 Compiler and library support for half-precision floating-point numbers 4-163
4.48 Half-precision floating-point number format 4-164
4.49 Compiler support for floating-point computations and linkage 4-165
4.50 Types of floating-point linkage 4-166
4.51 Compiler options for floating-point linkage and computations 4-167
4.52 Floating-point linkage and computational requirements of compiler options 4-169
4.53 Processors and their implicit Floating-Point Units (FPUs) 4-171

4.54 Integer division-by-zero errors in C code 4-173
4.55 Software floating-point division-by-zero errors in C code 4-175
4.56 About trapping software floating-point division-by-zero errors 4-176
4.57 Identification of software floating-point division-by-zero errors 4-177
4.58 Software floating-point division-by-zero debugging 4-179
4.59 New language features of C99 4-180
4.60 New library features of C99 4-182
4.61 // comments in C99 and C90 4-183
4.62 Compound literals in C99 4-184
4.63 Designated initializers in C99 4-185
4.64 Hexadecimal floating-point numbers in C99 4-186
4.65 Flexible array members in C99 4-187
4.66 __func__ predefined identifier in C99 4-188
4.67 inline functions in C99 4-189
4.68 long long data type in C99 and C90 4-190
4.69 Macros with a variable number of arguments in C99 4-191
4.70 Mixed declarations and statements in C99 4-192
4.71 New block scopes for selection and iteration statements in C99 4-193
4.72 _Pragma preprocessing operator in C99 4-194
4.73 Restricted pointers in C99 4-195
4.74 Additional <math.h> library functions in C99 4-196
4.75 Complex numbers in C99 4-197
4.76 Boolean type and <stdbool.h> in C99 4-198
4.77 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99 4-199
4.78 <fenv.h> floating-point environment access in C99 4-200
4.79 <stdio.h> snprintf family of functions in C99 4-201
4.80 <tgmath.h> type-generic math macros in C99 4-202
4.81 <wchar.h> wide character I/O functions in C99 4-203
4.82 How to prevent uninitialized data from being initialized to zero 4-204

Chapter 5 **Compiler Diagnostic Messages**
5.1 Severity of compiler diagnostic messages 5-206
5.2 Options that change the severity of compiler diagnostic messages 5-207
5.3 Controlling compiler diagnostic messages with pragmas 5-209
5.4 Prefix letters in compiler diagnostic messages 5-211
5.5 Compiler exit status codes and termination messages 5-212
5.6 Compiler data flow warnings 5-213

Chapter 6 **Using the Inline and Embedded Assemblers of the ARM Compiler**
6.1 Compiler support for inline assembly language 6-216
6.2 Inline assembler support in the compiler 6-217
6.3 Restrictions on inline assembler support in the compiler 6-218
6.4 Inline assembly language syntax with the __asm keyword in C and C++ 6-219
6.5 Inline assembly language syntax with the asm keyword in C++ 6-220
6.6 Inline assembler rules for compiler keywords __asm and asm 6-221
6.7 Restrictions on inline assembly operations in C and C++ code 6-222
6.8 Inline assembler register restrictions in C and C++ code 6-223
6.9 Inline assembler processor mode restrictions in C and C++ code 6-224
6.10 Inline assembler Thumb instruction set restrictions in C and C++ code 6-225
6.11 Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code 6-226

6.12 Inline assembler instruction restrictions in C and C++ code 6-227
6.13 Miscellaneous inline assembler restrictions in C and C++ code 6-228
6.14 Inline assembler and register access in C and C++ code 6-229
6.15 Inline assembler and the # constant expression specifier in C and C++ code 6-231
6.16 Inline assembler and instruction expansion in C and C++ code 6-232
6.17 Expansion of inline assembler instructions that use constants 6-233
6.18 Expansion of inline assembler load and store instructions 6-234
6.19 Inline assembler effect on processor condition flags in C and C++ code 6-235
6.20 Inline assembler expression operands in C and C++ code 6-236
6.21 Inline assembler register list operands in C and C++ code 6-237
6.22 Inline assembler intermediate operands in C and C++ code 6-238
6.23 Inline assembler function calls and branches in C and C++ code 6-239
6.24 Inline assembler branches and labels in C and C++ code 6-241
6.25 Inline assembler and virtual registers 6-242
6.26 Embedded assembler support in the compiler 6-243
6.27 Embedded assembler syntax in C and C++ 6-244
6.28 Effect of compiler ARM and Thumb states on embedded assembler 6-245
6.29 Restrictions on embedded assembly language functions in C and C++ code 6-246
6.30 Compiler generation of embedded assembly language functions 6-247
6.31 Access to C and C++ compile-time constant expressions from embedded assembler 6-249
6.32 Differences between expressions in embedded assembler and C or C++ 6-250
6.33 Manual overload resolution in embedded assembler 6-251
6.34 __offsetof_base keyword for related base classes in embedded assembler 6-252
6.35 Compiler-supported keywords for calling class member functions in embedded assembler 6-253
6.36 __mcall_is_virtual(D, f) 6-254
6.37 __mcall_is_in_vbase(D, f) 6-255
6.38 __mcall_offsetof_vbase(D, f) 6-256
6.39 __mcall_this_offset(D, f) 6-257
6.40 __vcall_offsetof_vfunc(D, f) 6-258
6.41 Calling nonstatic member functions in embedded assembler 6-259
6.42 Calling a nonvirtual member function 6-260
6.43 Calling a virtual member function 6-261
6.44 Accessing sp (r13), lr (r14), and pc (r15) 6-262
6.45 Differences in compiler support for inline and embedded assembly code 6-263

Chapter 7 **Compiler Command-line Options**

7.1 -Aopt 7-268
7.2 --allow_fpreg_for_nonfpdata, --no_allow_fpreg_for_nonfpdata 7-269
7.3 --allow_null_this, --no_allow_null_this 7-270
7.4 --alternative_tokens, --no_alternative_tokens 7-271
7.5 --anachronisms, --no_anachronisms 7-272
7.6 --apcs=qualifier...qualifier 7-273
7.7 --arm 7-277
7.8 --arm_only 7-278
7.9 --asm 7-279
7.10 --asm_dir=directory_name 7-280
7.11 --autoinline, --no_autoinline 7-281
7.12 --bigend 7-282

7.13 --bitband 7-283
7.14 --branch_tables, --no_branch_tables 7-284
7.15 --brief_diagnostics, --no_brief_diagnostics 7-286
7.16 --bss_threshold=num 7-287
7.17 -c 7-288
7.18 -C 7-289
7.19 --c90 7-290
7.20 --c99 7-291
7.21 --code_gen, --no_code_gen 7-292
7.22 --compatible=name 7-293
7.23 --compile_all_input, --no_compile_all_input 7-295
7.24 --conditionalize, --no_conditionalize 7-296
7.25 --cpp 7-297
7.26 --cpp11 7-298
7.27 --cpp_compat 7-299
7.28 --cpu=list 7-301
7.29 --cpu=name compiler option 7-302
7.30 --create_pch=filename 7-304
7.31 -Dname[(parm-list)][=def] 7-305
7.32 --data_reorder, --no_data_reorder 7-306
7.33 --debug, --no_debug 7-307
7.34 --debug_macros, --no_debug_macros 7-308
7.35 --default_extension=ext 7-309
7.36 --dep_name, --no_dep_name 7-310
7.37 --depend=filename 7-311
7.38 --depend_dir=directory_name 7-312
7.39 --depend_format=string 7-313
7.40 --depend_single_line, --no_depend_single_line 7-314
7.41 --depend_system_headers, --no_depend_system_headers 7-315
7.42 --depend_target=target 7-316
7.43 --diag_error=tag[,tag,...] 7-317
7.44 --diag_remark=tag[,tag,...] 7-318
7.45 --diag_style=arm|ide|gnu compiler option 7-319
7.46 --diag_suppress=tag[,tag,...] 7-320
7.47 --diag_suppress=optimizations 7-321
7.48 --diag_warning=tag[,tag,...] 7-322
7.49 --diag_warning=optimizations 7-323
7.50 --dollar, --no_dollar 7-324
7.51 --dwarf2 7-325
7.52 --dwarf3 7-326
7.53 -E 7-327
7.54 --echo 7-328
7.55 --emit_frame_directives, --no_emit_frame_directives 7-329
7.56 --enum_is_int 7-330
7.57 --errors=filename 7-331
7.58 --exceptions, --no_exceptions 7-332
7.59 --exceptions_unwind, --no_exceptions_unwind 7-333
7.60 --execute_only 7-334
7.61 --extended_initializers, --no_extended_initializers 7-335
7.62 --feedback=filename 7-336

7.63 --float_literal_pools, --no_float_literal_pools 7-337
7.64 --force_new_nothrow, --no_force_new_nothrow 7-338
7.65 --forceinline 7-339
7.66 --fp16_format=format 7-340
7.67 --fpmode=model 7-341
7.68 --fpu=list 7-343
7.69 --fpu=name compiler option 7-344
7.70 --friend_injection, --no_friend_injection 7-347
7.71 -g 7-348
7.72 --global_reg=reg_name[,reg_name,...] 7-349
7.73 --gnu 7-350
7.74 --gnu_defaults 7-351
7.75 --gnu_instrument, --no_gnu_instrument 7-352
7.76 --gnu_version=version 7-353
7.77 --guiding_decls, --no_guiding_decls 7-354
7.78 --help 7-355
7.79 -Idir[,dir,...] 7-356
7.80 --ignore_missing_headers 7-357
7.81 --implicit_include, --no_implicit_include 7-358
7.82 --implicit_include_searches, --no_implicit_include_searches 7-359
7.83 --implicit_key_function, --no_implicit_key_function 7-360
7.84 --implicit_typename, --no_implicit_typename 7-361
7.85 --info=totals 7-362
7.86 --inline, --no_inline 7-363
7.87 --integer_literal_pools, --no_integer_literal_pools 7-364
7.88 --interface_enums_are_32_bit 7-365
7.89 --interleave 7-366
7.90 -Jdir[,dir,...] 7-367
7.91 --kandr_include 7-368
7.92 -Lopt 7-369
7.93 --library_interface=lib 7-370
7.94 --library_type=lib 7-372
7.95 --liclinger=seconds 7-373
7.96 --licretry 7-374
7.97 --link_all_input, --no_link_all_input 7-375
7.98 --list 7-376
7.99 --list_dir=directory_name 7-378
7.100 --list_macros 7-379
7.101 --littleend 7-380
7.102 --locale=lang_country 7-381
7.103 --long_long 7-382
7.104 --loop_optimization_level=opt 7-383
7.105 --loose_implicit_cast 7-384
7.106 --lower_ropi, --no_lower_ropi 7-385
7.107 --lower_rwpi, --no_lower_rwpi 7-386
7.108 -M 7-387
7.109 --md 7-388
7.110 --message_locale=lang_country[.codepage] 7-389
7.111 --min_array_alignment=opt 7-390
7.112 --mm 7-391

7.113 --multibyte_chars, --no_multibyte_chars 7-392
7.114 --multifile, --no_multifile 7-393
7.115 --multiply_latency=cycles 7-394
7.116 --narrow_volatile_bitfields 7-395
7.117 --nonstd_qualifier_deduction, --no_nonstd_qualifier_deduction 7-396
7.118 -o filename 7-397
7.119 -Onum 7-399
7.120 --old_specializations, --no_old_specializations 7-402
7.121 --old_style_preprocessing 7-403
7.122 --omf_browse 7-404
7.123 --ool_section_name, --no_ool_section_name 7-405
7.124 -Ospace 7-406
7.125 -Otime 7-407
7.126 --output_dir=directory_name 7-408
7.127 -P 7-409
7.128 --parse_templates, --no_parse_templates 7-410
7.129 --pch 7-411
7.130 --pch_dir=dir 7-412
7.131 --pch_messages, --no_pch_messages 7-413
7.132 --pch_verbose, --no_pch_verbose 7-414
7.133 --pending_instantiations=n 7-415
7.134 --phony_targets 7-416
7.135 --pointer_alignment=num 7-417
7.136 --preinclude=filename 7-418
7.137 --preprocess_assembly 7-419
7.138 --preprocessed 7-420
7.139 --protect_stack, --no_protect_stack 7-421
7.140 --reassociate_saturation, --no_reassociate_saturation 7-422
7.141 --reduce_paths, --no_reduce_paths 7-423
7.142 --relaxed_ref_def, --no_relaxed_ref_def 7-424
7.143 --remarks 7-425
7.144 --remove_unneeded_entities, --no_remove_unneeded_entities 7-426
7.145 --restrict, --no_restrict 7-427
7.146 --retain=option 7-428
7.147 --rtti, --no_rtti 7-429
7.148 --rtti_data, --no_rtti_data 7-430
7.149 -S 7-431
7.150 --share_inlineable_strings, --no_share_inlineable_strings 7-432
7.151 --show_cmdline 7-434
7.152 --signed_bitfields, --unsigned_bitfields 7-435
7.153 --signed_chars, --unsigned_chars 7-436
7.154 --split_ldm 7-437
7.155 --split_sections 7-438
7.156 --strict, --no_strict 7-439
7.157 --strict_warnings 7-440
7.158 --string_literal_pools, --no_string_literal_pools 7-441
7.159 --sys_include 7-443
7.160 --thumb 7-444
7.161 --trigraphs, --no_trigraphs 7-445
7.162 --type_traits_helpers, --no_type_traits_helpers 7-446

7.163 -Uname 7-447
7.164 --unaligned_access, --no_unaligned_access 7-448
7.165 --use_frame_pointer, --no_use_frame_pointer 7-450
7.166 --use_pch=filename 7-451
7.167 --using_std, --no_using_std 7-452
7.168 --version_number 7-453
7.169 --vfe, --no_vfe 7-454
7.170 --via=filename 7-455
7.171 --vla, --no_vla 7-456
7.172 --vsn 7-457
7.173 -W 7-458
7.174 --wchar, --no_wchar 7-459
7.175 --wchar16 7-460
7.176 --wchar32 7-461
7.177 --whole_program 7-462
7.178 --wrap_diagnostics, --no_wrap_diagnostics 7-463

Chapter 8 Language Extensions
8.1 Preprocessor extensions 8-466
8.2 #assert 8-467
8.3 #include_next 8-468
8.4 #unassert 8-469
8.5 #warning 8-470
8.6 C99 language features available in C90 8-471
8.7 // comments 8-472
8.8 Subscripting struct 8-473
8.9 Flexible array members 8-474
8.10 C99 language features available in C++ and C90 8-475
8.11 Variadic macros 8-476
8.12 long long 8-477
8.13 restrict 8-478
8.14 Hexadecimal floats 8-479
8.15 Standard C language extensions 8-480
8.16 Constant expressions 8-481
8.17 Array and pointer extensions 8-482
8.18 Block scope function declarations 8-483
8.19 Dollar signs in identifiers 8-484
8.20 Top-level declarations 8-485
8.21 Benign redeclarations 8-486
8.22 External entities 8-487
8.23 Function prototypes 8-488
8.24 Standard C++ language extensions 8-489
8.25 ? operator 8-490
8.26 Declaration of a class member 8-491
8.27 friend 8-492
8.28 Read/write constants 8-493
8.29 Scalar type constants 8-494
8.30 Specialization of nonmember function templates 8-495
8.31 Type conversions 8-496
8.32 Standard C and Standard C++ language extensions 8-497

8.33 Address of a register variable 8-498
8.34 Arguments to functions 8-499
8.35 Anonymous classes, structures and unions 8-500
8.36 Assembler labels 8-501
8.37 Empty declaration 8-502
8.38 Hexadecimal floating-point constants 8-503
8.39 Incomplete enums 8-504
8.40 Integral type extensions 8-505
8.41 Label definitions 8-506
8.42 Long float 8-507
8.43 Nonstatic local variables 8-508
8.44 Structure, union, enum, and bitfield extensions 8-509
8.45 GNU extensions to the C and C++ languages 8-510

Chapter 9 **Compiler-specific Features**

9.1 Keywords and operators 9-515
9.2 __align 9-516
9.3 __ALIGNOF__ 9-517
9.4 __alignof__ 9-518
9.5 __asm 9-519
9.6 __forceinline 9-520
9.7 __global_reg 9-521
9.8 __inline 9-523
9.9 __int64 9-524
9.10 __INTADDR__ 9-525
9.11 __irq 9-526
9.12 __packed 9-527
9.13 __pure 9-529
9.14 __smc 9-530
9.15 __softfp 9-531
9.16 __svc 9-532
9.17 __svc_indirect 9-533
9.18 __svc_indirect_r7 9-534
9.19 __value_in_regs 9-535
9.20 __weak 9-536
9.21 __writeonly 9-538
9.22 __declspec attributes 9-539
9.23 __declspec(noinline) 9-540
9.24 __declspec(noreturn) 9-541
9.25 __declspec(nothrow) 9-542
9.26 __declspec(notshared) 9-543
9.27 __declspec(thread) 9-544
9.28 Function attributes 9-545
9.29 __attribute__((alias)) function attribute 9-547
9.30 __attribute__((always_inline)) function attribute 9-549
9.31 __attribute__((const)) function attribute 9-550
9.32 __attribute__((constructor[(priority)])) function attribute 9-551
9.33 __attribute__((deprecated)) function attribute 9-552
9.34 __attribute__((destructor[(priority)])) function attribute 9-553
9.35 __attribute__((format)) function attribute 9-554

9.36 __attribute__((format_arg(string-index))) function attribute 9-555
9.37 __attribute__((malloc)) function attribute 9-556
9.38 __attribute__((noinline)) function attribute 9-557
9.39 __attribute__((no_instrument_function)) function attribute 9-558
9.40 __attribute__((nomerge)) function attribute 9-559
9.41 __attribute__((nonnull)) function attribute 9-560
9.42 __attribute__((noreturn)) function attribute 9-561
9.43 __attribute__((notailcall)) function attribute 9-562
9.44 __attribute__((nothrow)) function attribute 9-563
9.45 __attribute__((pcs("calling_convention"))) function attribute 9-564
9.46 __attribute__((pure)) function attribute 9-565
9.47 __attribute__((section("name"))) function attribute 9-566
9.48 __attribute__((sentinel)) function attribute 9-567
9.49 __attribute__((unused)) function attribute 9-568
9.50 __attribute__((used)) function attribute 9-569
9.51 __attribute__((visibility("visibility_type"))) function attribute 9-570
9.52 __attribute__((warn_unused_result)) 9-571
9.53 __attribute__((weak)) function attribute 9-572
9.54 __attribute__((weakref("target"))) function attribute 9-573
9.55 Type attributes 9-574
9.56 __attribute__((bitband)) type attribute 9-575
9.57 __attribute__((aligned)) type attribute 9-576
9.58 __attribute__((packed)) type attribute 9-577
9.59 __attribute__((transparent_union)) type attribute 9-578
9.60 Variable attributes 9-579
9.61 __attribute__((alias)) variable attribute 9-580
9.62 __attribute__((at(address))) variable attribute 9-581
9.63 __attribute__((aligned)) variable attribute 9-582
9.64 __attribute__((deprecated)) variable attribute 9-583
9.65 __attribute__((noinline)) constant variable attribute 9-584
9.66 __attribute__((packed)) variable attribute 9-585
9.67 __attribute__((section("name"))) variable attribute 9-586
9.68 __attribute__((unused)) variable attribute 9-587
9.69 __attribute__((used)) variable attribute 9-588
9.70 __attribute__((visibility("visibility_type"))) variable attribute 9-589
9.71 __attribute__((weak)) variable attribute 9-590
9.72 __attribute__((weakref("target"))) variable attribute 9-591
9.73 __attribute__((zero_init)) variable attribute 9-592
9.74 Pragmas 9-593
9.75 #pragma anon_unions, #pragma no_anon_unions 9-594
9.76 #pragma arm 9-595
9.77 #pragma arm section [section_type_list] 9-596
9.78 #pragma diag_default tag[,tag,...] 9-598
9.79 #pragma diag_error tag[,tag,...] 9-599
9.80 #pragma diag_remark tag[,tag,...] 9-600
9.81 #pragma diag_suppress tag[,tag,...] 9-601
9.82 #pragma diag_warning tag[, tag, ...] 9-602
9.83 #pragma exceptions_unwind, #pragma no_exceptions_unwind 9-603
9.84 #pragma GCC system_header 9-604
9.85 #pragma hdrstop 9-605

9.86 #pragma import symbol_name 9-606
9.87 #pragma import(__use_full_stdio) 9-607
9.88 #pragma import(__use_smaller_memcpy) 9-608
9.89 #pragma inline, #pragma no_inline 9-609
9.90 #pragma no_pch 9-610
9.91 #pragma Onum 9-611
9.92 #pragma once 9-612
9.93 #pragma Ospace 9-613
9.94 #pragma Otime 9-614
9.95 #pragma pack(n) 9-615
9.96 #pragma pop 9-617
9.97 #pragma push 9-618
9.98 #pragma softfp_linkage, #pragma no_softfp_linkage 9-619
9.99 #pragma thumb 9-620
9.100 #pragma unroll [(n)] 9-621
9.101 #pragma unroll_completely 9-623
9.102 #pragma weak symbol, #pragma weak symbol1 = symbol2 9-624
9.103 Instruction intrinsics 9-625
9.104 __breakpoint intrinsic 9-626
9.105 __cdp intrinsic 9-627
9.106 __clrex intrinsic 9-628
9.107 __clz intrinsic 9-629
9.108 __current_pc intrinsic 9-630
9.109 __current_sp intrinsic 9-631
9.110 __disable_fiq intrinsic 9-632
9.111 __disable_irq intrinsic 9-633
9.112 __dmb intrinsic 9-635
9.113 __dsb intrinsic 9-636
9.114 __enable_fiq intrinsic 9-637
9.115 __enable_irq intrinsic 9-638
9.116 __fabs intrinsic 9-639
9.117 __fabsf intrinsic 9-640
9.118 __force_loads intrinsic 9-641
9.119 __force_stores intrinsic 9-642
9.120 __isb intrinsic 9-643
9.121 __ldrex intrinsic 9-644
9.122 __ldrexd intrinsic 9-646
9.123 __ldrt intrinsic 9-647
9.124 __memory_changed intrinsic 9-648
9.125 __nop intrinsic 9-649
9.126 __pld intrinsic 9-651
9.127 __pli intrinsic 9-652
9.128 __promise intrinsic 9-653
9.129 __qadd intrinsic 9-654
9.130 __qdbl intrinsic 9-655
9.131 __qsub intrinsic 9-656
9.132 __rbit intrinsic 9-657
9.133 __rev intrinsic 9-658
9.134 __return_address intrinsic 9-659
9.135 __ror intrinsic 9-660

9.136 __schedule_barrier intrinsic ... 9-661
9.137 __semihost intrinsic ... 9-662
9.138 __sev intrinsic ... 9-664
9.139 __sqrt intrinsic ... 9-665
9.140 __sqrtf intrinsic ... 9-666
9.141 __ssat intrinsic ... 9-667
9.142 __strex intrinsic ... 9-668
9.143 __strexd intrinsic ... 9-670
9.144 __strt intrinsic ... 9-672
9.145 __swp intrinsic ... 9-673
9.146 __usat intrinsic ... 9-674
9.147 __wfe intrinsic ... 9-675
9.148 __wfi intrinsic ... 9-676
9.149 __yield intrinsic ... 9-677
9.150 ARMv6 SIMD intrinsics ... 9-678
9.151 ETSI basic operations ... 9-679
9.152 C55x intrinsics ... 9-681
9.153 VFP status intrinsic ... 9-682
9.154 __vfp_status intrinsic ... 9-683
9.155 Fused Multiply Add (FMA) intrinsics ... 9-684
9.156 Named register variables ... 9-685
9.157 GNU built-in functions ... 9-689
9.158 Predefined macros ... 9-697
9.159 Built-in function name variables ... 9-703

Chapter 10 **C and C++ Implementation Details**
10.1 Character sets and identifiers in ARM C and C++ ... 10-705
10.2 Basic data types in ARM C and C++ ... 10-707
10.3 Operations on basic data types ARM C and C++ ... 10-709
10.4 Structures, unions, enumerations, and bitfields in ARM C and C++ ... 10-710
10.5 Using the ::operator new function in ARM C++ ... 10-715
10.6 Tentative arrays in ARM C++ ... 10-716
10.7 Old-style C parameters in ARM C++ functions ... 10-717
10.8 Anachronisms in ARM C++ ... 10-718
10.9 Template instantiation in ARM C++ ... 10-719
10.10 Namespaces in ARM C++ ... 10-720
10.11 C++ exception handling in ARM C++ ... 10-722
10.12 Extern inline functions in ARM C++ ... 10-723
10.13 C++11 supported features ... 10-724

Chapter 11 **What is Semihosting?**
11.1 What is semihosting? ... 11-729
11.2 The semihosting interface ... 11-730
11.3 Can I change the semihosting operation numbers? ... 11-731
11.4 Debug agent interaction SVCs ... 11-732
11.5 angel_SWIreason_EnterSVC (0x17) ... 11-733
11.6 angel_SWIreason_ReportException (0x18) ... 11-734
11.7 SYS_CLOSE (0x02) ... 11-736
11.8 SYS_CLOCK (0x10) ... 11-737
11.9 SYS_ELAPSED (0x30) ... 11-738

11.10 SYS_ERRNO (0x13) 11-739
11.11 SYS_FLEN (0x0C) 11-740
11.12 SYS_GET_CMDLINE (0x15) 11-741
11.13 SYS_HEAPINFO (0x16) 11-742
11.14 SYS_ISERROR (0x08) 11-743
11.15 SYS_ISTTY (0x09) 11-744
11.16 SYS_OPEN (0x01) 11-745
11.17 SYS_READ (0x06) 11-746
11.18 SYS_READC (0x07) 11-747
11.19 SYS_REMOVE (0x0E) 11-748
11.20 SYS_RENAME (0x0F) 11-749
11.21 SYS_SEEK (0x0A) 11-750
11.22 SYS_SYSTEM (0x12) 11-751
11.23 SYS_TICKFREQ (0x31) 11-752
11.24 SYS_TIME (0x11) 11-753
11.25 SYS_TMPNAM (0x0D) 11-754
11.26 SYS_WRITE (0x05) 11-755
11.27 SYS_WRITEC (0x03) 11-756
11.28 SYS_WRITE0 (0x04) 11-757

Chapter 12 **ARMv6 SIMD Instruction Intrinsics**

12.1 ARMv6 SIMD intrinsics by prefix 12-760
12.2 ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags 12-762
12.3 ARMv6 SIMD intrinsics, compatible processors and architectures 12-765
12.4 ARMv6 SIMD instruction intrinsics and APSR GE flags 12-766
12.5 __qadd16 intrinsic 12-768
12.6 __qadd8 intrinsic 12-769
12.7 __qasx intrinsic 12-770
12.8 __qsax intrinsic 12-771
12.9 __qsub16 intrinsic 12-772
12.10 __qsub8 intrinsic 12-773
12.11 __sadd16 intrinsic 12-774
12.12 __sadd8 intrinsic 12-775
12.13 __sasx intrinsic 12-776
12.14 __sel intrinsic 12-777
12.15 __shadd16 intrinsic 12-778
12.16 __shadd8 intrinsic 12-779
12.17 __shasx intrinsic 12-780
12.18 __shsax intrinsic 12-781
12.19 __shsub16 intrinsic 12-782
12.20 __shsub8 intrinsic 12-783
12.21 __smlad intrinsic 12-784
12.22 __smladx intrinsic 12-785
12.23 __smlald intrinsic 12-786
12.24 __smlaldx intrinsic 12-787
12.25 __smlsd intrinsic 12-788
12.26 __smlsdx intrinsic 12-789
12.27 __smlsld intrinsic 12-790
12.28 __smlsldx intrinsic 12-791
12.29 __smuad intrinsic 12-792

12.30 __smuadx intrinsic 12-793
12.31 __smusd intrinsic 12-794
12.32 __smusdx intrinsic 12-795
12.33 __ssat16 intrinsic 12-796
12.34 __ssax intrinsic 12-797
12.35 __ssub16 intrinsic 12-798
12.36 __ssub8 intrinsic 12-799
12.37 __sxtab16 intrinsic 12-800
12.38 __sxtb16 intrinsic 12-801
12.39 __uadd16 intrinsic 12-802
12.40 __uadd8 intrinsic 12-803
12.41 __uasx intrinsic 12-804
12.42 __uhadd16 intrinsic 12-805
12.43 __uhadd8 intrinsic 12-806
12.44 __uhasx intrinsic 12-807
12.45 __uhsax intrinsic 12-808
12.46 __uhsub16 intrinsic 12-809
12.47 __uhsub8 intrinsic 12-810
12.48 __uqadd16 intrinsic 12-811
12.49 __uqadd8 intrinsic 12-812
12.50 __uqasx intrinsic 12-813
12.51 __uqsax intrinsic 12-814
12.52 __uqsub16 intrinsic 12-815
12.53 __uqsub8 intrinsic 12-816
12.54 __usad8 intrinsic 12-817
12.55 __usada8 intrinsic 12-818
12.56 __usat16 intrinsic 12-819
12.57 __usax intrinsic 12-820
12.58 __usub16 intrinsic 12-821
12.59 __usub8 intrinsic 12-822
12.60 __uxtab16 intrinsic 12-823
12.61 __uxtb16 intrinsic 12-824

Chapter 13 Via File Syntax
13.1 Overview of via files 13-826
13.2 Via file syntax rules 13-827

Chapter 14 Summary Table of GNU Language Extensions
14.1 Supported GNU extensions 14-829

Chapter 15 Standard C Implementation Definition
15.1 Implementation definition 15-833
15.2 Translation 15-834
15.3 Environment 15-835
15.4 Identifiers 15-837
15.5 Characters 15-838
15.6 Integers 15-840
15.7 Floating-point 15-841
15.8 Arrays and pointers 15-842
15.9 Registers 15-843

15.10 Structures, unions, enumerations, and bitfields 15-844
15.11 Qualifiers 15-848
15.12 Expression evaluation 15-849
15.13 Preprocessing directives 15-850
15.14 Library functions 15-851
15.15 Behaviors considered undefined by the ISO C Standard 15-852

Chapter 16 **Standard C++ Implementation Definition**

16.1 Integral conversion 16-854
16.2 Calling a pure virtual function 16-855
16.3 Major features of language support 16-856
16.4 Standard C++ library implementation definition 16-857

Chapter 17 **C and C++ Compiler Implementation Limits**

17.1 C++ ISO/IEC standard limits 17-859
17.2 Limits for integral numbers 17-861
17.3 Limits for floating-point numbers 17-862

List of Figures
ARM® Compiler v5.06 for µVision® armcc User Guide

Figure 4-1	*Half-precision floating-point format*	*4-164*
Figure 9-1	*Nonpacked structure S*	*9-615*
Figure 9-2	*Packed structure SP*	*9-615*
Figure 10-1	*Conventional nonpacked structure example*	*10-711*
Figure 10-2	*Bitfield allocation 1*	*10-712*
Figure 10-3	*Bitfield allocation 2*	*10-713*
Figure 11-1	*Semihosting overview*	*11-729*
Figure 15-1	*Conventional nonpacked structure example*	*15-845*
Figure 15-2	*Bitfield allocation 1*	*15-847*
Figure 15-3	*Bitfield allocation 2*	*15-847*

List of Tables
ARM® Compiler v5.06 for µVision® armcc User Guide

Table 2-1	Filename suffixes recognized by the compiler	2-47
Table 2-2	Include file search paths	2-51
Table 4-1	C code for incrementing and decrementing loops	4-113
Table 4-2	C Disassembly for incrementing and decrementing loops	4-113
Table 4-3	C code for rolled and unrolled bit-counting loops	4-115
Table 4-4	Disassembly for rolled and unrolled bit-counting loops	4-116
Table 4-5	C code for nonvolatile and volatile buffer loops	4-117
Table 4-6	Disassembly for nonvolatile and volatile buffer loop	4-118
Table 4-7	C code for pure and impure functions	4-129
Table 4-8	Disassembly for pure and impure functions	4-129
Table 4-9	Compiler storage of data objects by byte alignment	4-144
Table 4-10	C code for an unpacked struct, a packed struct, and a struct with individually packed fields	4-152
Table 4-11	Disassembly for an unpacked struct, a packed struct, and a struct with individually packed fields	4-152
Table 4-12	C code for a packed struct and a pragma packed struct	4-153
Table 4-13	Compiler options for floating-point linkage and floating-point computations	4-167
Table 4-14	FPU-option capabilities and requirements	4-169
Table 4-15	Implicit FPUs of processors	4-171
Table 5-1	Severity of diagnostic messages	5-206
Table 5-2	Identifying diagnostic messages	5-211
Table 6-1	Differences between inline and embedded assembler	6-263

Table 7-1 Compiling with the --asm option 7-279
Table 7-2 Compatible processor or architecture combinations 7-293
Table 7-3 Compiling with the --interleave option 7-366
Table 7-4 Compiling with the -o option 7-397
Table 7-5 Compiling without the -o option 7-397
Table 8-1 Behavior of constant value initializers in comparison with ISO Standard C 8-481
Table 9-1 Keyword extensions that the ARM compiler supports 9-515
Table 9-2 __declspec attributes that the compiler supports, and their equivalents 9-539
Table 9-3 Function attributes that the compiler supports, and their equivalents 9-545
Table 9-4 Type attributes that the compiler supports, and their equivalents 9-574
Table 9-5 Variable attributes that the compiler supports, and their equivalents 9-579
Table 9-6 Pragmas that the compiler supports 9-593
Table 9-7 Instruction intrinsics that the ARM compiler supports 9-625
Table 9-8 Access widths that the __ldrex intrinsic supports 9-644
Table 9-9 Access widths that the __ldrex intrinsic supports 9-646
Table 9-10 Access widths that the __ldrt intrinsic supports 9-647
Table 9-11 Access widths that the __strex intrinsic supports 9-668
Table 9-12 Access widths that the __strexd intrinsic supports 9-670
Table 9-13 Access widths that the __strt intrinsic supports 9-672
Table 9-14 Access widths that the __swp intrinsic supports 9-673
Table 9-15 ETSI basic operations that the ARM compilation tools support 9-679
Table 9-16 ETSI status flags exposed in the ARM compilation tools 9-679
Table 9-17 TI C55x intrinsics that the compilation tools support 9-681
Table 9-18 Modifying the FPSCR flags 9-683
Table 9-19 Named registers available on ARM architecture-based processors 9-686
Table 9-20 Named registers available on targets with floating-point hardware 9-686
Table 9-21 Predefined macros 9-697
Table 9-22 Thumb architecture versions in relation to ARM architecture versions 9-702
Table 9-23 built-in variables 9-703
Table 10-1 Character escape codes 10-705
Table 10-2 Size and alignment of data types 10-707
Table 11-1 Hardware vector reason codes 11-734
Table 11-2 Software reason codes 11-734
Table 11-3 Value of mode 11-745
Table 12-1 ARMv6 SIMD intrinsics by prefix 12-760
Table 12-2 ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags 12-762
Table 12-3 ARMv6 SIMD intrinsics, compatible processors and architectures 12-765
Table 12-4 ARMv6 SIMD instruction intrinsics and APSR GE flags 12-766
Table 14-1 Supported GNU extensions 14-829
Table 15-1 Character escape codes 15-838
Table 16-1 Major feature support for language 16-856
Table 17-1 Implementation limits 17-859
Table 17-2 Integer ranges 17-861
Table 17-3 Floating-point limits 17-862
Table 17-4 Other floating-point characteristics 17-862

Preface

This preface introduces the *ARM® Compiler v5.06 for µVision® armcc User Guide*.

It contains the following:

• *About this book* on page 24.

About this book

ARM® Compiler v5.05 for µVision® armcc User Guide. This manual provides user information for the ARM compiler, `armcc`. `armcc` is an optimizing C and C++ compiler that compiles Standard C and Standard C++ source code into machine code for ARM architecture-based processors. Available as PDF.

Using this book

This book is organized into the following chapters:

Chapter 1 Overview of the Compiler
Gives an overview of the ARM compiler, the languages and extensions it supports, and the provided libraries.

Chapter 2 Getting Started with the Compiler
Introduces some of the more common ARM compiler command-line options.

Chapter 3 Compiler Features
Provides an overview of ARM-specific features of the compiler.

Chapter 4 Compiler Coding Practices
Describes programming techniques and practices to help you increase the portability, efficiency and robustness of your C and C++ source code.

Chapter 5 Compiler Diagnostic Messages
Describes the format of compiler diagnostic messages and how to control the output during compilation.

Chapter 6 Using the Inline and Embedded Assemblers of the ARM Compiler
Describes the optimizing inline assembler and non-optimizing embedded assembler of the ARM compiler, `armcc`.

Chapter 7 Compiler Command-line Options
Describes the `armcc` compiler command-line options.

Chapter 8 Language Extensions
Describes the language extensions that the compiler supports.

Chapter 9 Compiler-specific Features
Describes compiler-specific features including ARM extensions to the C and C++ Standards, ARM-specific pragmas and intrinsics, and predefined macros.

Chapter 10 C and C++ Implementation Details
Describes the language implementation details for the compiler. Some language implementation details are common to both C and C++, while others are specific to C++.

Chapter 11 What is Semihosting?
Describes the semihosting mechanism.

Chapter 12 ARMv6 SIMD Instruction Intrinsics
Describes the ARMv6 SIMD instruction intrinsics. SIMD instructions allow the processor to operate on packed 8-bit or 16-bit values in 32-bit registers.

Chapter 13 Via File Syntax
Describes the syntax of via files accepted by the `armcc`.

Chapter 14 Summary Table of GNU Language Extensions
Describes ARM compiler support for GNU extensions to the C and C++ languages.

Chapter 15 Standard C Implementation Definition
Provides information required by the ISO C standard for conforming C implementations.

Chapter 16 Standard C++ Implementation Definition

Lists the C++ language features defined in the ISO/IEC standard for C++, and states whether or not ARM C++ supports that language feature.

Chapter 17 C and C++ Compiler Implementation Limits

Describes the implementation limits when using the ARM compiler to compile C and C++.

Glossary

The ARM Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the *ARM Glossary* for more information.

Typographic conventions

italic

Introduces special terminology, denotes cross-references, and citations.

bold

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace` (underlined: mono)

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

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

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *ARM® Compiler v5.06 for µVision® armcc User Guide*.
- The number ARM DUI0375G_02.

- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- *ARM Information Center*.
- *ARM Technical Support Knowledge Articles*.
- *Support and Maintenance*.
- *ARM Glossary*.

Chapter 1
Overview of the Compiler

Gives an overview of the ARM compiler, the languages and extensions it supports, and the provided libraries.

It contains the following sections:

- *1.1 The compiler* on page 1-28.
- *1.2 Source language modes of the compiler* on page 1-29.
- *1.3 Language extensions* on page 1-31.
- *1.4 Language compliance* on page 1-32.
- *1.5 The C and C++ libraries* on page 1-33.

1.1 The compiler

The compiler, `armcc`, is an optimizing C and C++ compiler that compiles Standard C and Standard C++ source code into machine code for ARM architecture-based processors.

Command-line options enable you to control the level of optimization.

The compiler compiles the following different varieties of C and C++ source code into ARM and Thumb® code:

- ISO Standard C:1990 source.
- ISO Standard C:1999 source.
- ISO Standard C++:2003 source.
- ISO Standard C++:2011 source.

Publications on the C and C++ standards are available from national standards bodies. For example, AFNOR in France and ANSI in the USA.

`armcc` complies with the *Base Standard Application Binary Interface for the ARM Architecture* (BSABI). In particular, the compiler:

- Generates output objects in ELF format.
- Generates *Debug With Arbitrary Record Format* Debugging Standard Version 3 (DWARF 3) debug information and contains support for DWARF 2 debug tables.
- Uses the *Edison Design Group* (EDG) front end.

Many features of the compiler are designed to take advantage of the target processor or architecture that your code is designed to run on, so knowledge of your target processor or architecture is useful, and in some cases, essential, when working with the compiler.

Note

Be aware of the following:

- Generated code might be different between two ARM® Compiler releases.
- For a feature release, there might be significant code generation differences.

Note

The command-line option descriptions and related information in the individual ARM Compiler tools documents describe all the features that ARM Compiler supports. Any features not documented are not supported and are used at your own risk. You are responsible for making sure that any generated code using unsupported features is operating correctly.

Related information

The DWARF Debugging Standard, http://dwarfstd.org/.

Application Binary Interface (ABI) for the ARM Architecture.

1.2 Source language modes of the compiler

The compiler can compile different varieties of C and C++ source code.

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`.

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.

ISO C++03

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

- ISO/IEC 14882:2003. The 2003 International Standard for C++.

ISO C++11

The compiler compiles supported features of C++11 as defined by the 2011 standard.

- ISO/IEC 14882:2011. The 2011 International Standard for C++.

The compiler provides support for numerous extensions to the C and C++ languages. For example, it supports some GNU compiler extensions. The compiler has several modes in which compliance with 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.

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.

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

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.

Use the compiler options `--C90 --strict` to enforce strict C90 code. Because `C90` is the default, you could omit `--C90`.

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.

Use the compiler options `--c99 --strict` to compile strict C99 code.

Standard C

Means C90 or C99 as appropriate.

C

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

C++03

Means ISO C++03, excepting wide streams and export templates, either with or without the ARM extensions.

Use the compiler option `--cpp` to compile C++03 code.

Use the compiler options `--cpp --cpp_compat` to maximize binary compatibility with C++03 code compiled using older compiler versions.

strict C++03

Means ISO C++03, excepting wide streams and export templates.

Use the compiler options `--cpp --strict` to compile strict C++03 code.

C++11

Means ISO C++11, excepting wide streams and export templates, either with or without the ARM extensions.

Use the compiler option `--cpp11` to compile C++11 code.

Use the compiler options `--cpp11 --cpp_compat` to compile a subset of C++11 code that maximizes compatibility with code compiled to the C++ 2003 standard.

strict C++11

Means ISO C++11, excepting wide streams and export templates.

Use the compiler options `--cpp11 --strict` to compile strict C++11 code.

Standard C++

Means strict C++03 or strict C++11 as appropriate.

C++

Means any of C++03, strict C++03, C++11, strict C++11.

Related concepts

4.59 New language features of C99 on page 4-180.

4.64 Hexadecimal floating-point numbers in C99 on page 4-186.

Related references

1.3 Language extensions on page 1-31.

1.4 Language compliance on page 1-32.

1.3 Language extensions on page 1-31.

1.4 Language compliance on page 1-32.

15.1 Implementation definition on page 15-833.

16.4 Standard C++ library implementation definition on page 16-857.

1.3 Language extensions

The compiler supports numerous extensions to its various source languages.

These language extensions 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.

Standard C extensions

The compiler supports numerous extensions to strict C99, for example, function prototypes that override old-style nonprototype definitions.

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.

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.

GNU extensions

The compiler supports some GNU extensions.

ARM-specific extensions

The compiler supports a range of extensions specific to the ARM compiler, for example, instruction intrinsics and other built-in functions.

Related references

8.6 C99 language features available in C90 on page 8-471.

8.10 C99 language features available in C++ and C90 on page 8-475.

8.15 Standard C language extensions on page 8-480.

8.24 Standard C++ language extensions on page 8-489.

8.32 Standard C and Standard C++ language extensions on page 8-497.

1.4 Language compliance on page 1-32.

Chapter 14 Summary Table of GNU Language Extensions on page 14-828.

1.4 Language compliance

The compiler provides several command-line options for either enforcing or relaxing compliance with the available source languages.

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.

———————————

Examples

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++03.
- 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.

Related references

7.73 --gnu on page 7-350.

7.156 --strict, --no_strict on page 7-439.

8.45 GNU extensions to the C and C++ languages on page 8-510.

2.7 Filename suffixes recognized by the compiler on page 2-47.

Chapter 14 Summary Table of GNU Language Extensions on page 14-828.

1.5 The C and C++ libraries

ARM provides a number of runtime C and C++ libraries, including the ARM C libraries, the Rogue Wave Standard C++ Library, and ARM 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 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).

Rogue Wave Standard C++ Library

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. These manuals might be installed with the documentation of your ARM product. If they are not installed, you can view them at *Rogue Wave Standard C++ Library Documentation*

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.

Related information

ARM DS-5 License Management Guide.

Application Binary Interface (ABI) for the ARM Architecture.

Compliance with the Application Binary Interface (ABI) for the ARM architecture.

The ARM C and C++ Libraries.

Chapter 2
Getting Started with the Compiler

Introduces some of the more common ARM compiler command-line options.

It contains the following sections:

- *2.1 Compiler command-line syntax* on page 2-35.
- *2.2 Compiler command-line options listed by group* on page 2-36.
- *2.3 Default compiler behavior* on page 2-42.
- *2.4 Order of compiler command-line options* on page 2-43.
- *2.5 Using stdin to input source code to the compiler* on page 2-44.
- *2.6 Directing output to stdout* on page 2-46.
- *2.7 Filename suffixes recognized by the compiler* on page 2-47.
- *2.8 Compiler output files* on page 2-49.
- *2.9 Factors influencing how the compiler searches for header files* on page 2-50.
- *2.10 Compiler command-line options and search paths* on page 2-51.
- *2.11 Compiler search rules and the current place* on page 2-52.
- *2.12 The ARMCC5INC environment variable* on page 2-53.
- *2.13 Code compatibility between separately compiled and assembled modules* on page 2-54.
- *2.14 Linker feedback during compilation* on page 2-55.
- *2.15 Unused function code* on page 2-56.
- *2.16 Minimizing code size by eliminating unused functions during compilation* on page 2-57.
- *2.17 Compilation build time* on page 2-58.

2.1 Compiler command-line syntax

Use the `armcc` command from the command-line to invoke the compiler. Specify the source files you want to compile, together with any options you need to control compiler behavior.

The command for invoking the compiler is:

`armcc [options] [source]`

where:

`options`
: are compiler command-line options that affect the behavior of the compiler.

`source`
: provides the filenames of one or more text files containing C or C++ source code. By default, the compiler looks for source files and creates output files in the current directory.

  If a source file is an assembly file, that is, one with an extension of `.s`, the compiler activates the ARM assembler to process the source file.

  When you invoke the compiler, you normally specify one or more source files. However, a minority of compiler command-line options do not require you to specify a source file. For example, `armcc --version_number`.

The compiler accepts one or more input files, for example:

```
armcc -c [options] input_file_1 ... input_file_n
```

Specifying a dash `-` for an input file causes the compiler to read from `stdin`. To specify that all subsequent arguments are treated as filenames, not as command switches, use the POSIX option `--`.

The `-c` option instructs the compiler to perform the compilation step, but not the link step.

Related concepts

2.2 Compiler command-line options listed by group on page 2-36.

Related references

7.17 -c on page 7-288.

Related information

Rules for specifying command-line options.

Toolchain environment variables.

2.2 Compiler command-line options listed by group

This topic lists the compiler command-line options, ordered by functional group.

Note

The following characters are interchangeable:

- Nonprefix hyphens and underscores. For example, `--version_number` and `--version-number`.
- Equals signs and spaces. For example, `armcc --cpu=list` and `armcc --cpu list`.

This applies to all tools provided with the compiler.

The compiler command-line options are as follows:

Help

- `--echo`
- `--help`
- `--show_cmdline`
- `--version_number`
- `--vsn`

Source languages

- `--c90`
- `--c99`
- `--compile_all_input, --no_compile_all_input`
- `--cpp`
- `--cpp11`
- `--cpp_compat`
- `--gnu`
- `--strict, --no_strict`
- `--strict_warnings`

Search paths

- `-Idir[,dir,...]`
- `-Jdir[,dir,...]`
- `--kandr_include`
- `--preinclude=filename`
- `--reduce_paths, --no_reduce_paths`
- `--sys_include`
- `--ignore_missing_headers`

Precompiled headers

- `--create_pch=filename`
- `--pch`
- `--pch_dir=dir`
- `--pch_messages, --no_pch_messages`
- `--pch_verbose, --no_pch_verbose`
- `--use_pch=filename`

Preprocessor

- `-C`
- `--code_gen, --no_code_gen`
- `-Dname[(parm-list)][=def]`
- `-E`
- `-M`
- `--old_style_preprocessing`
- `-P`
- `--preprocess_assembly`
- `--preprocessed`
- `-Uname`

C++

- `--allow_null_this`
- `--anachronisms, --no_anachronisms`
- `--dep_name, --no_dep_name`
- `--force_new_nothrow, --no_force_new_nothrow`
- `--friend_injection, --no_friend_injection`
- `--guiding_decls, --no_guiding_decls`
- `--implicit_include, --no_implicit_include`
- `--implicit_include_searches, --no_implicit_include_searches`
- `--implicit_typename, --no_implicit_typename`
- `--nonstd_qualifier_deduction, --no_nonstd_qualifier_deduction`
- `--old_specializations, --no_old_specializations`
- `--parse_templates, --no_parse_templates`
- `--pending_instantiations=n`
- `--rtti, --no_rtti`
- `--rtti_data`
- `--type_traits_helpers`
- `--using_std, --no_using_std`
- `--vfe, --no_vfe`

Output format

- `--asm`
- `--asm_dir`
- `-c`
- `--default_extension=ext`
- `--depend=filename`
- `--depend_dir`
- `--depend_format=string`
- `--depend_single_line`
- `--depend_system_headers, --no_depend_system_headers`
- `--depend_target`
- `--errors`
- `--info=totals`
- `--interleave`
- `--list`
- `--list_dir`
- `--list_macros`
- `--md`
- `--mm`
- `-o filename`
- `--output_dir`
- `--phony_targets`
- `-S`
- `--split_sections`

Target architectures and processors

- `--arm`
- `--arm_only`
- `--compatible=name`
- `--cpu=list`
- `--cpu=name`
- `--fpu=list`
- `--fpu=name`
- `--thumb`

Floating-point support

- `--fp16_format=format`
- `--fpmode=model`
- `--fpu=list`
- `--fpu=name`

Debug

- `--debug, --no_debug`
- `--debug_macros, --no_debug_macros`
- `--dwarf2`
- `--dwarf3`
- `-g`
- `--remove_unneeded_entities, --no_remove_unneeded_entities`
- `--emit_frame_directives`

Code generation

- `--allow_fpreg_for_nonfpdata, --no_allow_fpreg_for_nonfpdata`
- `--alternative_tokens, --no_alternative_tokens`
- `--bigend`
- `--bitband`
- `--branch_tables`
- `--bss_threshold=num`
- `--conditionalize, --no_conditionalize`
- `--default_definition_visibility`
- `--dollar, --no_dollar`
- `--enum_is_int`
- `--exceptions, --no_exceptions`
- `--exceptions_unwind, --no_exceptions_unwind`
- `--execute_only`
- `--float_literal_pools`
- `--export_defs_implicitly, --no_export_defs_implicitly`
- `--extended_initializers, --no_extended_initializers`
- `--global_reg`
- `--gnu_defaults`
- `--gnu_instrument`
- `--gnu_version`
- `--implicit_key_function`
- `--integer_literal_pools`
- `--interface_enums_are_32_bit`
- `--littleend`
- `--locale=lang_country`
- `--long_long`
- `--loose_implicit_cast`
- `--message_locale=lang_country[.codepage]`
- `--min_array_alignment=opt`
- `--multibyte_chars, --no_multibyte_chars`
- `--multiply_latency`
- `--narrow_volatile_bitfields`
- `--pointer_alignment=num`
- `--protect_stack, --no_protect_stack`
- `--restrict, --no_restrict`
- `--relaxed_ref_def`
- `--share_inlineable_strings`
- `--signed_bitfields, --unsigned_bitfields`
- `--signed_chars, --unsigned_chars`
- `--split_ldm`
- `--string_literal_pools`
- `--trigraphs`
- `--unaligned_access, --no_unaligned_access`
- `--use_frame_pointer`
- `--vla, --no_vla`
- `--wchar`
- `--wchar16`
- `--wchar32`

Optimization

- `--autoinline, --no_autoinline`
- `--data_reorder, --no_data_reorder`
- `--forceinline`
- `--fpmode=model`
- `--inline, --no_inline`
- `--library_interface=lib`
- `--library_type=lib`
- `--loop_optimization_level=opt`
- `--lower_ropi, --no_lower_ropi`
- `--lower_rwpi, --no_lower_rwpi`
- `--multifile, --no_multifile`
- `-Onum`
- `-Ospace`
- `-Otime`
- `--reassociate_saturation`
- `--retain=option`
- `--whole_program`

Note

Optimization options can limit the debug information generated by the compiler.

Diagnostics

- `--brief_diagnostics, --no_brief_diagnostics`
- `--diag_error=tag[,tag,...]`
- `--diag_remark=tag[,tag,...]`
- `--diag_style={arm|ide|gnu}`
- `--diag_suppress=tag[,tag,...]`
- `--diag_suppress=optimizations`
- `--diag_warning=tag[,tag,...]`
- `--diag_warning=optimizations`
- `--errors=filename`
- `--link_all_input`
- `--remarks`
- `-W`
- `--wrap_diagnostics, --no_wrap_diagnostics`

Command-line options in a text file

- `--via=filename`

Linker feedback

- `--feedback=filename`

Procedure call standard

- `--apcs=qualifier...qualifier`

Licensing

- `--liclinger`
- `--licretry`

Passing options to other tools

- `-Aopt`
- `-Lopt`

Other options

- `--omf_browse`

Related concepts

2.4 Order of compiler command-line options on page 2-43.

Related references

Chapter 7 Compiler Command-line Options on page 7-264.

2.3 Default compiler behavior

By default, the compiler determines the source language by examining the source filename extension. For example, `filename.c` indicates C, while `filename.cpp` indicates C++03, although the command-line options `--c90`, `--c99`, `--cpp`, and `--cpp11` let you override this.

The default compiler target instruction set depends on the target processor (`--cpu=name`):

- For processors that support ARM instructions, the default instruction set is ARM. Use the `--thumb` command-line option to specify Thumb®.
- For processors that do not support ARM instructions, the default instruction set is Thumb.

When you compile multiple files with a single command, all files must be of the same type, either C or C++. The compiler cannot switch the language based on the file extension. The following example produces an error because the specified source files have different languages:

```
armcc -c test1.c test2.cpp
```

If you specify files with conflicting file extensions you can force the compiler to compile both files for C or for C++, regardless of file extension. For example:

```
armcc -c --cpp test1.c test2.cpp
```

Where an unrecognized extension begins with `.c`, for example, *`filename`*`.cmd`, an error message is generated.

Support for processing *Precompiled Header* (PCH) files is not available when you specify multiple source files in a single compilation. If you request PCH processing and specify more than one primary source file, the compiler issues an error message, and aborts the compilation.

Note

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

`armcc` can in turn invoke `armasm` and `armlink`. For example, if your source code contains embedded assembly code, `armasm` is called. `armcc` searches for the `armasm` and `armlink` binaries in the following locations, in this order:

1. The same location as `armcc`.
2. The `PATH` locations.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
2.4 Order of compiler command-line options on page 2-43.
2.9 Factors influencing how the compiler searches for header files on page 2-50.
2.11 Compiler search rules and the current place on page 2-52.
2.12 The ARMCC5INC environment variable on page 2-53.
2.2 Compiler command-line options listed by group on page 2-36.
2.1 Compiler command-line syntax on page 2-35.

Related tasks

2.5 Using stdin to input source code to the compiler on page 2-44.

Related references

2.7 Filename suffixes recognized by the compiler on page 2-47.
2.8 Compiler output files on page 2-49.

2.4 Order of compiler command-line options

In general, compiler command-line options can appear in any order in a single compiler invocation. However, the effects of some options depend on the order they appear in the command line and how they are combined with other related options.

The compiler enables you to use multiple options even where these might conflict. This means that you can append new options to an existing command line, for example, in a makefile or a via file.

Where options override previous options on the same command line, the last option specified always takes precedence. For example:

```
armcc -O1 -O2 -Ospace -Otime ...
```

is executed by the compiler as:

```
armcc -O2 -Otime
```

You can use the environment variable `ARMCC5_CCOPT` to specify compiler command-line options. Options specified on the command line take precedence over options specified in the environment variable.

To see how the compiler has processed the command line, use the `--show_cmdline` option. This shows nondefault options that the compiler used. The contents of any via files are expanded. In the example used here, although the compiler executes `armcc -O2 -Otime`, the output from `--show_cmdline` does not include `-O2`. This is because `-O2` is the default optimization level, and `--show_cmdline` does not show options that apply by default.

Related concepts

2.2 Compiler command-line options listed by group on page 2-36.

2.5 Using stdin to input source code to the compiler

Instead of creating a file for your source code, you can use `stdin` to input source code directly on the command line.

This is useful if you want to test a short piece of code without having to create a file for it.

Procedure

1. Invoke the compiler with the command-line options you want to use. The default compiler mode is C. Use the minus character (`-`) as the source filename to instruct the compiler to take input from `stdin`. For example:

   ```
   armcc --bigend -c -
   ```

   If you want an object file to be written, use the `-o` option. If you want preprocessor output to be sent to the output stream, use the `-E` option. If you want the output to be sent to `stdout`, use the `-o-` option. If you want an assembly listing of the keyboard input to be sent to the output stream after input has been terminated, use none of these options.
2. You cannot input on the same line after the minus character. You must press the return key if you have not already done so.

   The command prompt waits for you to enter more input.
3. Enter your input. For example:

   ```
   #include <stdio.h>
   int main(void)
   { printf("Hello world\n"); }
   ```
4. Terminate your input by entering:
   - `Ctrl+Z` then `Return` on Microsoft Windows systems.
   - `Ctrl+D` on Red Hat Linux systems.

An assembly listing for the keyboard input is sent to the output stream after input has been terminated if both the following are true:

- No output file is specified.
- No preprocessor-only option is specified, for example `-E`.

Otherwise, an object file is created or preprocessor output is sent to the standard output stream, depending on whether you used the `-o` option or the `-E` option.

The compiler accepts source code from the standard input stream in combination with other files, when performing a link step. For example, the following are permitted:

- `armcc -o output.axf - object.o mylibrary.a`
- `armcc -o output.axf --c90 source.c -`

Executing the following command compiles the source code you provide on standard input, and links it into `test.axf`:

```
armcc -o test.axf -
```

You can only combine standard input with other source files when you are linking code. If you attempt to combine standard input with other source files when not linking, the compiler generates an error.

Related concepts

2.1 Compiler command-line syntax on page 2-35.

2.2 Compiler command-line options listed by group on page 2-36.

Related information

Rules for specifying command-line options.

Toolchain environment variables.

Rules for specifying command-line options.

Toolchain environment variables.

2.6 Directing output to stdout

If you want output to be sent to the standard output stream, use the `-o-` option.

For example:

```
armcc -c -o- hello.c
```

This outputs an assembly listing of the source code to `stdout`.

To send preprocessor output to `stdout`, use the `-E` option.

Related concepts

2.1 Compiler command-line syntax on page 2-35.

2.2 Compiler command-line options listed by group on page 2-36.

Related information

Rules for specifying command-line options.

Toolchain environment variables.

2.7 Filename suffixes recognized by the compiler

The compiler uses filename suffixes to identify the classes of file involved in compilation and in the link stage.

The filename suffixes recognized by the compiler are described in the following table.

Note

Explicitly specifying `--c90`, `--c99`, `--cpp`, or `--cpp11` overrides the effect of filename suffixes.

Table 2-1 Filename suffixes recognized by the compiler

Suffix	Description	Usage notes
`.c`	C source file	Implies `--c90`
`.C`	C or C++ source file	Implies `--c90`.

`.cpp`
`.c++`
`.cxx`
`.cc`
`.CC`
	C++ source file	Implies `--cpp`
		The compiler uses the suffixes `.cc` and `.CC` to identify files for implicit inclusion.

`.d`	Dependency list file	`.d` is the default output filename suffix for files output using the `--md` option.
`.h`	C or C++ header file	-
`.i`	C or C++ source file	A C or C++ file that has already been preprocessed, and is to be compiled without additional preprocessing.
`.ii`	C++ source file	A C++ file that has already been preprocessed, and is to be compiled without additional preprocessing.
`.lst`	Error and warning list file	`.lst` is the default output filename suffix for files output using the `--list` option.

`.a`
`.lib`
`.o`
`.obj`
`.so`
	ARM, Thumb, or mixed ARM and Thumb object file or library.	-

`.pch`	Precompiled header file	`.pch` is the default output filename suffix for files output using the `--pch` option. **Note** Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.
`.s`	ARM, Thumb, or mixed ARM and Thumb assembly language source file.	For files in the input file list suffixed with `.s`, the compiler invokes the assembler, `armasm`, to assemble the file. `.s` is the default output filename suffix for files output using either the option `-S` or `--asm`.

Table 2-1 Filename suffixes recognized by the compiler (continued)

Suffix	Description	Usage notes
`.S`	ARM, Thumb, or mixed ARM and Thumb assembly language source file.	`.S` is equivalent to `.s`.
`.sx`	ARM, Thumb, or mixed ARM and Thumb assembly language source file.	For files in the input file list suffixed with `.sx`, the compiler preprocesses the assembly source before passing that source to the assembler.
`.txt`	Text file	`.txt` is the default output filename suffix for files output using the `-S` or `--asm` option in combination with the `--interleave` option.

Related references

7.7 --arm on page 7-277.
7.89 --interleave on page 7-366.
7.98 --list on page 7-376.
7.109 --md on page 7-388.
7.129 --pch on page 7-411.
7.149 -S on page 7-431.
7.23 --compile_all_input, --no_compile_all_input on page 7-295.
10.9 Template instantiation in ARM C++ on page 10-719.

2.8 Compiler output files

By default, output files created by the compiler are located in the current directory. Object files are written in ARM ELF.

Related information

ELF for the ARM Architecture.

2.9 Factors influencing how the compiler searches for header files

Several factors influence how the compiler searches for `#include` header files and source files.

- The value of the environment variable `ARMCC5INC`.
- The value of the environment variable `ARMINC`.
- The `-I` and `-J` compiler options.
- The `--kandr_include` and `--sys_include` compiler options.
- Whether the filename is an absolute filename or a relative filename.
- Whether the filename is between angle brackets or double quotes.

Related concepts

2.12 The ARMCC5INC environment variable on page 2-53.

2.11 Compiler search rules and the current place on page 2-52.

Related references

2.10 Compiler command-line options and search paths on page 2-51.

7.79 -Idir[,dir,...] on page 7-356.

7.90 -Jdir[,dir,...] on page 7-367.

7.91 --kandr_include on page 7-368.

7.159 --sys_include on page 7-443.

Related information

Toolchain environment variables.

2.10 Compiler command-line options and search paths

The following table shows how the specified compiler command-line options affect the search path used by the compiler when it searches for header and source files.

Table 2-2 Include file search paths

Compiler option	<include> search order	"include" search order
Neither `-Idir[,dir,...]` nor `-Jdir[,dir,...]`	1. `ARMCC5INC` 2. `ARMINC` 3. `../include`	1. The *current place* on page 2-52. 2. `ARMCC5INC` 3. `ARMINC` 4. `../include`
`-Idir[,dir,...]`	1. `ARMCC5INC` 2. `ARMINC` 3. `../include` 4. The directory or directories specified by `-Idir[,dir,...]` .	1. The *current place* on page 2-52. 2. The directory or directories specified by -`Idir[,dir,...]`. 3. `ARMCC5INC` 4. `ARMINC` 5. `../include`
`-Jdir[,dir,...]`	The directory or directories specified by -`Jdir[,dir,...]`.	1. The *current place* on page 2-52. 2. The directory or directories specified by -`Jdir[,dir,...]`.
Both `-Idir[,dir,...]` and `-Jdir[,dir,...]`	1. The directory or directories specified by `-Jdir[,dir,...]`. 2. The directory or directories specified by `-Idir[,dir,...]`.	1. The *current place* on page 2-52. 2. The directory or directories specified by -`Idir[,dir,...]`. 3. The directory or directories specified by -`Jdir[,dir,...]`.
`--sys_include`	No effect.	Removes the *current place* on page 2-52 from the search path.
`--kandr_include`	No effect.	Uses Kernighan and Ritchie search rules.

Related concepts

2.12 The ARMCC5INC environment variable on page 2-53.
2.11 Compiler search rules and the current place on page 2-52.
2.9 Factors influencing how the compiler searches for header files on page 2-50.

Related references

7.79 -Idir[,dir,...] on page 7-356.
7.90 -Jdir[,dir,...] on page 7-367.
7.91 --kandr_include on page 7-368.
7.159 --sys_include on page 7-443.

2.11 Compiler search rules and the current place

By default, the compiler uses Berkeley UNIX search rules, so source files and `#include` header files are searched for relative to the *current place*. The current place is the directory containing the source or header file currently being processed by the compiler.

When a file is found relative to an element of the search path, the directory containing that file becomes the new current place. When the compiler has finished processing that file, it restores the previous current place. At each instant there is a stack of current places corresponding to the stack of nested `#include` directives. For example, if the current place is the include directory `...\include`, and the compiler is seeking the include file `sys\defs.h`, it locates `...\include\sys\defs.h` if it exists. When the compiler begins to process `defs.h`, the current place becomes `...\include\sys`. Any file included by `defs.h` that is not specified with an absolute path name, is searched for relative to `...\include\sys`.

The original current place `...\include` is restored only when the compiler has finished processing `defs.h`.

You can disable the stacking of current places by using the compiler option `--kandr_include`. This option makes the compiler use Kernighan and Ritchie search rules whereby each nonrooted user `#include` is searched for relative to the directory containing the source file that is being compiled.

Related concepts

2.12 The ARMCC5INC environment variable on page 2-53.

2.9 Factors influencing how the compiler searches for header files on page 2-50.

Related references

2.10 Compiler command-line options and search paths on page 2-51.

7.79 -Idir[,dir,...] on page 7-356.

7.90 -Jdir[,dir,...] on page 7-367.

7.91 --kandr_include on page 7-368.

7.159 --sys_include on page 7-443.

2.12 The ARMCC5INC environment variable

The `ARMCC5INC` environment variable points to the location of the included header and source files that are provided with the compilation tools.

This variable might be initialized with the correct path to the header files when the ARM compilation tools are installed or when configured with server modules. You can change this variable, but you must ensure that any changes you make do not break the installation.

The list of directories specified by the `ARMCC5INC` environment variable is semi-colon separated.

If you want to include files from other locations, use the `-I` and `-J` command-line options as required.

When compiling, directories specified with `ARMCC5INC` are searched immediately after directories specified by the `-I` option have been searched, for user include files.

If you use the `-J` option, `ARMCC5INC` is ignored.

Related concepts

2.11 Compiler search rules and the current place on page 2-52.

2.9 Factors influencing how the compiler searches for header files on page 2-50.

Related references

2.10 Compiler command-line options and search paths on page 2-51.

7.79 -Idir[,dir,...] on page 7-356.

7.90 -Jdir[,dir,...] on page 7-367.

7.91 --kandr_include on page 7-368.

7.159 --sys_include on page 7-443.

Related information

Toolchain environment variables.

2.13 Code compatibility between separately compiled and assembled modules

By writing code that adheres to the ARM Architecture Procedure Call Standard (AAPCS), you can ensure that separately compiled and assembled modules can work together.

The AAPCS forms part of the *Base Standard Application Binary Interface for the ARM Architecture* specification.

Interworking qualifiers associated with the `--apcs` compiler command-line option control interworking. Position independence qualifiers, also associated with the `--apcs` compiler command-line option, control position independence, and affect the creation of reentrant and thread-safe code.

Note

This does not mean that you must use the same `--apcs` command-line options to get your modules to work together. You must be familiar with the AAPCS.

Related references

7.6 --apcs=qualifier...qualifier on page 7-273.

Related information

Procedure Call Standard for the ARM Architecture.

ARM C libraries and multithreading.

2.14 Linker feedback during compilation

The compiler can use feedback files produced by the linker to optimize code generation.

Feedback from the linker to the compiler enables:
- Efficient elimination of unused functions.
- Reduction of compilation required for interworking.

Related concepts

2.15 Unused function code on page 2-56.

Related tasks

2.16 Minimizing code size by eliminating unused functions during compilation on page 2-57.

2.15 Unused function code

Unused function code can unnecessarily increase code size. Feedback from the linker to the compiler can remove unused function code, minimizing code size.

Unused function code might occur in the following situations.

- Where you have legacy functions that are no longer used in your source code. Rather than manually remove the unused function code from your source code, you can use linker feedback to remove the unused object code automatically from the final image.
- Where a function is inlined. Where an inlined function is not declared as **`static`**, the out-of-line function code is still present in the object file, but there is no longer a call to that code.

In addition, the linker can detect when an ARM function is being called from a Thumb state, and when a Thumb function is being called from an ARM state. You can use feedback from the linker to avoid compiling functions for interworking that are never used in an interworking context.

———— **Note** ————

Reduction of compilation required for interworking is only applicable to ARMv4T architectures. ARMv5T and later processors can interwork without penalty.

————————————

The linker option `--feedback=`*`filename`* creates a feedback file, and the `--feedback_type` option controls the different types of feedback generated.

Related tasks

2.16 Minimizing code size by eliminating unused functions during compilation on page 2-57.

Related references

2.14 Linker feedback during compilation on page 2-55.

2.16 Minimizing code size by eliminating unused functions during compilation

Feedback from the linker to the compiler enables efficient elimination of unused functions.

Procedure

1. Compile your source code.
2. Use the linker option `--feedback=`*`filename`* to create a feedback file.
3. Use the linker option `--feedback_type` to control which feedback the linker generates.

   By default, the linker generates feedback to eliminate unused functions. This is equivalent to `--feedback_type=unused,noiw`. The linker can also generate feedback to avoid compiling functions for interworking that are never used in an interworking context. Use the linker option `--feedback_type=unused,iw` to eliminate both types of unused function.

   Note

   Reduction of compilation required for interworking is only applicable to ARMv4T architectures. ARMv5T and later processors can interwork without penalty.

4. Re-compile using the compiler option `--feedback=`*`filename`* to feed the feedback file to the compiler.

The compiler uses the feedback file generated by the linker to compile the source code in a way that enables the linker to subsequently discard the unused functions.

Note

To obtain maximum benefit from linker feedback, do a full compile and link at least twice. A single compile and link using feedback from a previous build is normally sufficient to obtain some benefit.

Note

Always ensure that you perform a full clean build immediately before using the linker feedback file. This minimizes the risk of the feedback file becoming out of date with the source code it was generated from.

You can specify the `--feedback=`*`filename`* option even when no feedback file exists. This enables you to use the same build commands or makefile regardless of whether a feedback file exists, for example:

```
armcc -c --feedback=unused.txt test.c -o test.o
armlink --feedback=unused.txt test.o -o test.axf
```

The first time you build the application, it compiles normally but the compiler warns you that it cannot read the specified feedback file because it does not exist. The link command then creates the feedback file and builds the image. Each subsequent compilation step uses the feedback file from the previous link step to remove any unused functions that are identified.

Related concepts

2.15 Unused function code on page 2-56.

Related references

2.14 Linker feedback during compilation on page 2-55.
7.62 --feedback=filename on page 7-336.

Related information

--feedback_type=type linker option.
About linker feedback.

2.17 Compilation build time

Compilation build time is affected by the compiler optimizations you use and the applicaitons running on your host platform.

This section contains the following subsections:

- *2.17.1 Compilation build time* on page 2-58.
- *2.17.2 Minimizing compilation build time* on page 2-59.
- *2.17.3 Minimizing compilation build time with a single armcc invocation* on page 2-60.
- *2.17.4 Effect of --multifile on compilation build time* on page 2-60.
- *2.17.5 Minimizing compilation build time with parallel make* on page 2-61.
- *2.17.6 Compilation build time on Windows* on page 2-61.

2.17.1 Compilation build time

Modern software applications can comprise many thousands of source code files. These files can take a considerable amount of time to compile. The many different techniques that the ARM compilation tools use to optimize for small code size and high performance can also increase build time.

When you invoke the compiler, the following steps occur:

1. The compiler loads and begins to execute.
2. The compiler tries to obtain a license.
3. The compiler compiles your code.

Loading and beginning to execute the compiler normally takes a fixed period of time.

The time taken to obtain a license does not generally vary if a license is available. However, if a floating license is being used, the time taken to obtain a license depends on network traffic and whether or not a license is free on the server. In most cases, rather than terminate with error if a license is not immediately available, the compiler waits for a license to become available.

The process of obtaining a floating license is more involved than obtaining a node-locked license. With a node-locked license, the compiler only has to parse the file to check that there is a valid license. With a floating license, the compiler has to check where the license is, send a message through the TCP/IP stacks over the network to the server, then wait for a response. When the compiler receives the response, it then has to check whether or not it has been granted a license. When the compilation is complete, the license has to be returned back to the server.

Floating licenses provide flexibility, but at the cost of speed. If speed is your priority, consider obtaining node-locked licenses for your build machines, or some node-locked licenses locked to USB network cards that can be moved between machines as required.

Setting the environment variable `TCP_NODELAY` to `1` improves FlexNet license server system performance when processing license requests. However, you must use this with caution, because it might cause an increase in network traffic.

The time taken to compile your code depends on the size and complexity of the file being compiled. Compiling a small number of large files might be quicker than compiling a larger number of small files. This is because the longer compilation time per file might be offset by the smaller amount of time spent loading and unloading the compiler and obtaining licenses.

Related tasks

2.17.2 Minimizing compilation build time on page 2-59.

Related references

2.17.3 Minimizing compilation build time with a single armcc invocation on page 2-60.

2.17.4 Effect of --multifile on compilation build time on page 2-60.

2.17.5 Minimizing compilation build time with parallel make on page 2-61.

Compilation build time and operating system choice.

4.13 Methods of reducing debug information in objects and libraries on page 4-124.

Related information

Optimizing license checkouts from a floating license server.

Licensed features of ARM Compiler.

2.17.2 Minimizing compilation build time

There are a number of actions you can take to minimize how long the compiler takes to compile your source code.

These actions include:

- Avoid compiling at `-O3` level. `-O3` gives maximum optimization in the code that is generated, but can result in longer build times to achieve such results.
- Minimize the amount of debug information the compiler generates.
- Guard against multiple inclusion of header files.
- Use the **`restrict`** keyword if you can safely do so, to avoid the compiler having to do compile-time checks for pointer aliasing.
- Try to keep the number of include paths to a minimum. If you have many include paths, ensure that the files you include most often exist in directories near the start of the include search path.
- Try compiling a small number of large files instead of a large number of small files. The longer compilation time per file might be offset by less time spent unloading and unloading the compiler and obtaining licenses, particularly if using floating licenses.
- Try compiling multiple files within a single invocation of `armcc` (and single license checkout), instead of multiple `armcc` invocations.
- Floating licenses provide flexibility, but at the cost of speed. Consider obtaining node-locked licenses for your build machines, or some node-locked licenses locked to USB network cards that can be moved between machines as required.
- Consider using or avoiding `--multifile` compilation, depending on the resulting build time.

  Note

  — In RVCT 4.0, if you compile with `-O3`, `--multifile` is enabled by default.
  — In ARM Compiler 4.1 and later, `--multifile` is disabled by default, regardless of the optimization level.

- If you are using a makefile-based build environment, consider using a make tool that can apply some form of parallelism.
- Consider your choice of operating system for cross-compilation. Linux generally gives better build speed than Windows, but there are general performance-tuning techniques you can apply on Windows that might help improve build times.

Related concepts

2.17.1 Compilation build time on page 2-58.

3.21 Precompiled Header (PCH) files on page 3-88.

4.14 Guarding against multiple inclusion of header files on page 4-125.

Vectorization on loops containing pointers.

Related references

2.17.3 Minimizing compilation build time with a single armcc invocation on page 2-60.

2.17.4 Effect of --multifile on compilation build time on page 2-60.

2.17.5 Minimizing compilation build time with parallel make on page 2-61.

Compilation build time and operating system choice.

4.13 Methods of reducing debug information in objects and libraries on page 4-124.

7.30 --create_pch=filename on page 7-304.

7.114 --multifile, --no_multifile on page 7-393.
7.119 -Onum on page 7-399.
7.129 --pch on page 7-411.
7.130 --pch_dir=dir on page 7-412.
7.145 --restrict, --no_restrict on page 7-427.

Related information

Licensed features of ARM Compiler.

2.17.3 Minimizing compilation build time with a single armcc invocation

Using a single `armcc` invocation rather than multiple invocations helps minimize compilation build time.

The following type of script incurs multiple loads and unloads of the compiler and multiple license checkouts:

```
armcc file1.c ...
armcc file2.c ...
armcc file3.c ...
```

Instead, you can try modifying your script to compile multiple files within a single invocation of `armcc`. For example, `armcc` ***`file1.c file2.c file3.c`*** `...`

For convenience, you can also list all your `.c` files in a single via file invoked with `armcc -via` *`sources.txt`*.

Although this mechanism can dramatically reduce license checkouts and loading and unloading of the compiler to give significant improvements in build time, the following limitations apply:
- All files are compiled with the same options.
- Converting existing build systems could be difficult.
- Usability depends on source file structure and dependencies.
- An IDE might be unable to report which file had compilation errors.
- After detecting an error, the compiler does not compile subsequent files.

Related concepts

2.17.1 Compilation build time on page 2-58.

Related tasks

2.17.2 Minimizing compilation build time on page 2-59.

Related references

7.114 --multifile, --no_multifile on page 7-393.
7.119 -Onum on page 7-399.
7.170 --via=filename on page 7-455.

Related information

Licensed features of ARM Compiler.

2.17.4 Effect of --multifile on compilation build time

When compiling with `--multifile`, the compiler might generate code with additional optimizations by compiling across several source files to produce a single object file. These additional cross-source optimizations can increase compilation time.

Conversely, if there is little additional optimization to apply, and only small amounts of code to check for possible optimizations, then using `--multifile` to generate a single object file instead of several might reduce compilation time as a result of time recovered from creating (opening and closing) multiple object files.

Note

- In RVCT 4.0, if you compile with `-O3`, `--multifile` is enabled by default.
- In ARM Compiler 4.1 and later, `--multifile` is disabled by default, regardless of the optimization level.

Related concepts

2.17.1 Compilation build time on page 2-58.

Related tasks

2.17.2 Minimizing compilation build time on page 2-59.

Related references

7.114 --multifile, --no_multifile on page 7-393.

7.119 -Onum on page 7-399.

Related information

Licensed features of ARM Compiler.

2.17.5 Minimizing compilation build time with parallel make

If you are using a makefile-based build environment, you could consider using a make tool that can apply some form of parallelism to minimize compilation build time.

For example, with GNU `make` you can typically use `make -j N`, where `N` is the number of compile processes you want to have running in parallel.

Even on a single machine with a single processor, a performance boost can be achieved. This is because running processes in parallel can hide the effects of network delays and general I/O accesses such as loading and saving files to disk, by fully utilizing the processor during these times with another compilation process.

If you have multiple processor machines, you can extend the use of parallelism with `make -j N * M`, where `M` is the number of processors.

Related concepts

2.17.1 Compilation build time on page 2-58.

Related tasks

2.17.2 Minimizing compilation build time on page 2-59.

2.17.6 Compilation build time on Windows

There are ways to tune the performance of the Windows operating system at a general level. This might help with increasing the percentage of processor time that is being used for your build.

At a simple level, turning off virus checking software can help, but an Internet search for `"tune windows performance"` provides plenty of information.

Chapter 3
Compiler Features

Provides an overview of ARM-specific features of the compiler.

It contains the following sections:

- *3.1 Compiler intrinsics* on page 3-64.
- *3.2 Performance benefits of compiler intrinsics* on page 3-65.
- *3.3 ARM assembler instruction intrinsics* on page 3-66.
- *3.4 Generic intrinsics* on page 3-67.
- *3.5 Compiler intrinsics for controlling IRQ and FIQ interrupts* on page 3-68.
- *3.6 Compiler intrinsics for inserting optimization barriers* on page 3-69.
- *3.7 Compiler intrinsics for inserting native instructions* on page 3-70.
- *3.8 Compiler intrinsics for Digital Signal Processing (DSP)* on page 3-71.
- *3.9 Compiler support for European Telecommunications Standards Institute (ETSI) basic operations* on page 3-72.
- *3.10 Overflow and carry status flags for C and C++ code* on page 3-74.
- *3.11 Texas Instruments (TI) C55x intrinsics for optimizing C code* on page 3-75.
- *3.12 Compiler support for accessing registers using named register variables* on page 3-76.
- *3.13 Pragmas recognized by the compiler* on page 3-79.
- *3.14 Compiler and processor support for bit-banding* on page 3-81.
- *3.15 Compiler type attribute, __attribute__((bitband))* on page 3-82.
- *3.16 --bitband compiler command-line option* on page 3-83.
- *3.17 How the compiler handles bit-band objects placed outside bit-band regions* on page 3-84.
- *3.18 Compiler support for thread-local storage* on page 3-85.
- *3.19 Compiler support for literal pools* on page 3-86.
- *3.20 Compiler eight-byte alignment features* on page 3-87.
- *3.21 Precompiled Header (PCH) files* on page 3-88.
- *3.22 Automatic Precompiled Header (PCH) file processing* on page 3-90.

- *3.23 Precompiled Header (PCH) file processing and the header stop point* on page 3-91.
- *3.24 Precompiled Header (PCH) file creation requirements* on page 3-93.
- *3.25 Compilation with multiple Precompiled Header (PCH) files* on page 3-95.
- *3.26 Obsolete Precompiled Header (PCH) files* on page 3-96.
- *3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file* on page 3-97.
- *3.28 Selectively applying Precompiled Header (PCH) file processing* on page 3-98.
- *3.29 Suppressing Precompiled Header (PCH) file processing* on page 3-99.
- *3.30 Message output during Precompiled Header (PCH) processing* on page 3-100.
- *3.31 Performance issues with Precompiled Header (PCH) files* on page 3-101.
- *3.32 Default compiler options that are affected by optimization level* on page 3-102.

3.1 Compiler intrinsics

Compiler intrinsics are functions provided by the compiler. They enable you to easily incorporate domain-specific operations in C and C++ source code without resorting to complex implementations in assembly language.

The C and C++ languages are suited to a wide variety of tasks but they do not provide in-built support for specific areas of application, for example, *Digital Signal Processing* (DSP).

Within a given application domain, there is usually a range of domain-specific operations that have to be performed frequently. However, often these operations cannot be efficiently implemented in C or C++. A typical example is the saturated add of two 32-bit signed two's complement integers, commonly used in DSP programming. The following example shows a C implementation of saturated add operation

```
#include <limits.h>
int L_add(const int a, const int b)
{
    int c;
    c = a + b;
    if (((a ^ b) & INT_MIN) == 0)
    {
        if ((c ^ a) & INT_MIN)
        {
            c = (a < 0) ? INT_MIN : INT_MAX;
        }
    }
    return c;
}
```

Using compiler intrinsics, you can achieve more complete coverage of target architecture instructions than you would from the instruction selection of the compiler.

An intrinsic function has the appearance of a function call in C or C++, but is replaced during compilation by a specific sequence of low-level instructions. When implemented using an intrinsic, for example, the saturated add function previous example has the form:

```
#include <dspfns.h>    /* Include ETSI intrinsics */
...
int a, b, result;
...
result = L_add(a, b);  /* Saturated add of a and b */
```

Related concepts

3.5 Compiler intrinsics for controlling IRQ and FIQ interrupts on page 3-68.

3.9 Compiler support for European Telecommunications Standards Institute (ETSI) basic operations on page 3-72.

3.11 Texas Instruments (TI) C55x intrinsics for optimizing C code on page 3-75.

Related references

3.2 Performance benefits of compiler intrinsics on page 3-65.

3.3 ARM assembler instruction intrinsics on page 3-66.

9.151 ETSI basic operations on page 9-679.

9.103 Instruction intrinsics on page 9-625.

9.152 C55x intrinsics on page 9-681.

3.2 Performance benefits of compiler intrinsics

The use of compiler intrinsics offers a number of performance benefits:

- The low-level instructions substituted for an intrinsic might be more efficient than corresponding implementations in C or C++, resulting in both reduced instruction and cycle counts. To implement the intrinsic, the compiler automatically generates the best sequence of instructions for the specified target architecture. For example, the `L_add` intrinsic maps directly to the ARM assembly language instruction `qadd`:

```
QADD r0, r0, r1    /* Assuming r0 = a, r1 = b on entry */
```

- More information is given to the compiler than the underlying C and C++ language is able to convey. This enables the compiler to perform optimizations and to generate instruction sequences that it could not otherwise have performed.

These performance benefits can be significant for real-time processing applications. However, care is required because the use of intrinsics can decrease code portability.

Related concepts

3.1 Compiler intrinsics on page 3-64.

3.5 Compiler intrinsics for controlling IRQ and FIQ interrupts on page 3-68.

3.9 Compiler support for European Telecommunications Standards Institute (ETSI) basic operations on page 3-72.

3.11 Texas Instruments (TI) C55x intrinsics for optimizing C code on page 3-75.

3.3 ARM assembler instruction intrinsics

The compiler provides a range of instruction intrinsics for generating ARM assembly language instructions from within your C or C++ code.

Collectively, these intrinsics enable you to emulate inline assembly code using a combination of C code and instruction intrinsics.

ARM provides the following types of compiler intrinsics:

- Generic intrinsics.
- Compiler intrinsics for controlling IRQ and FIQ interrupts.
- Compiler intrinsics for inserting optimization barriers.
- Compiler intrinsics for inserting native instructions.
- Compiler intrinsics for Digital Signal Processing (DSP).

Related concepts

3.1 Compiler intrinsics on page 3-64.

3.5 Compiler intrinsics for controlling IRQ and FIQ interrupts on page 3-68.

3.9 Compiler support for European Telecommunications Standards Institute (ETSI) basic operations on page 3-72.

3.11 Texas Instruments (TI) C55x intrinsics for optimizing C code on page 3-75.

3.6 Compiler intrinsics for inserting optimization barriers on page 3-69.

3.8 Compiler intrinsics for Digital Signal Processing (DSP) on page 3-71.

Related references

3.4 Generic intrinsics on page 3-67.

3.7 Compiler intrinsics for inserting native instructions on page 3-70.

3.4 Generic intrinsics

The compiler provides a number of generic intrinsics, that is, intrinsics not targeted towards any particular area of application.

The following generic intrinsics are ARM language extensions to the ISO C and C++ standards:

- `__breakpoint` intrinsic.
- `__current_pc` intrinsic.
- `__current_sp` intrinsic.
- `__nop` intrinsic.
- `__return_address` intrinsic.
- `__semihost` intrinsic.

Implementations of these intrinsics are available across all architectures.

Related references

9.104 __breakpoint intrinsic on page 9-626.

9.108 __current_pc intrinsic on page 9-630.

9.109 __current_sp intrinsic on page 9-631.

9.125 __nop intrinsic on page 9-649.

9.134 __return_address intrinsic on page 9-659.

9.137 __semihost intrinsic on page 9-662.

3.5 Compiler intrinsics for controlling IRQ and FIQ interrupts

The intrinsics `__disable_irq`, `__enable_irq`, `__disable_fiq` and `__enable_fiq` control IRQ and FIQ interrupts.

You cannot use these intrinsics to change any other `CPSR` bits, including the mode, state, and imprecise data abort setting. This means that the intrinsics can be used only if the processor is already in a privileged mode, because the control bits of the `CPSR` and `SPSR` cannot be changed in User mode.

These intrinsics are available for all processor architectures in both ARM and Thumb state, as follows:

- If you are compiling for processors that support ARMv6 (or later), a `CPS` instruction is generated inline for these functions, for example:

  ```
      CPSID  i
  ```
- If you are compiling for processors that support ARMv4 or ARMv5 in ARM state, the compiler inlines a sequence of `MRS` and `MSR` instructions, for example:

  ```
      MRS  r0, CPSR
      ORR  r0, r0, #0x80
      MSR  CPSR_c, r0
  ```
- If you are compiling for processors that support ARMv4 or ARMv5 in Thumb state, or if `--compatible` is being used, the compiler calls a helper function, for example:

  ```
      BL    __ARM_disable_irq
  ```

Related concepts

3.1 Compiler intrinsics on page 3-64.

3.9 Compiler support for European Telecommunications Standards Institute (ETSI) basic operations on page 3-72.

3.11 Texas Instruments (TI) C55x intrinsics for optimizing C code on page 3-75.

Related references

3.2 Performance benefits of compiler intrinsics on page 3-65.

3.3 ARM assembler instruction intrinsics on page 3-66.

9.110 __disable_fiq intrinsic on page 9-632.

9.111 __disable_irq intrinsic on page 9-633.

9.114 __enable_fiq intrinsic on page 9-637.

9.115 __enable_irq intrinsic on page 9-638.

3.6 Compiler intrinsics for inserting optimization barriers

The optimization barrier intrinsics `__schedule_barrier`, `__force_stores`, `__force_loads`, and `__memory_changed` let you override compiler optimizations by disabling instruction re-ordering and forcing memory updates.

The compiler can perform a range of optimizations, including re-ordering instructions and merging some operations. In some cases, such as system level programming where memory is being accessed concurrently by multiple processes, it might be necessary to disable instruction re-ordering and force memory to be updated.

The optimization barrier intrinsics `__schedule_barrier`, `__force_stores`, `__force_loads` and `__memory_changed` do not generate code, but they can result in slightly increased code size and additional memory accesses.

Note

On some systems the memory barrier intrinsics might not be sufficient to ensure memory consistency. For example, the `__memory_changed` intrinsic forces values held in registers to be written out to memory. However, if the destination for the data is held in a region that can be buffered it might wait in a write buffer. In this case you might also have to write to CP15 or use a memory barrier instruction to drain the write buffer. See the Technical Reference Manual for your ARM processor for more information.

Related references

9.119 __force_stores intrinsic on page 9-642.
9.124 __memory_changed intrinsic on page 9-648.
9.136 __schedule_barrier intrinsic on page 9-661.
9.118 __force_loads intrinsic on page 9-641.

3.7 Compiler intrinsics for inserting native instructions

The compiler provides a number of intrinsics that insert ARM processor instructions into the instruction stream generated by the compiler.

Related references

9.105 __cdp intrinsic on page 9-627.
9.106 __clrex intrinsic on page 9-628.
9.121 __ldrex intrinsic on page 9-644.
9.123 __ldrt intrinsic on page 9-647.
9.126 __pld intrinsic on page 9-651.
9.127 __pli intrinsic on page 9-652.
9.132 __rbit intrinsic on page 9-657.
9.133 __rev intrinsic on page 9-658.
9.135 __ror intrinsic on page 9-660.
9.138 __sev intrinsic on page 9-664.
9.142 __strex intrinsic on page 9-668.
9.144 __strt intrinsic on page 9-672.
9.145 __swp intrinsic on page 9-673.
9.147 __wfe intrinsic on page 9-675.
9.148 __wfi intrinsic on page 9-676.
9.149 __yield intrinsic on page 9-677.

3.8 Compiler intrinsics for Digital Signal Processing (DSP)

The compiler provides intrinsics that assist in the implementation of DSP algorithms.

These intrinsics introduce the appropriate target instructions for:

- ARM, on architectures from ARMv5TE onwards.
- Thumb, on architectures with Thumb-2 technology.

Not every instruction has its own intrinsic. The compiler can combine several intrinsics, or combinations of intrinsics and C operators to generate more powerful instructions. For example, the ARMv5TE `QDADD` instruction is generated by a combination of `__qadd` and `__qdbl`.

Related references

9.107 __clz intrinsic on page 9-629.
9.116 __fabs intrinsic on page 9-639.
9.117 __fabsf intrinsic on page 9-640.
9.129 __qadd intrinsic on page 9-654.
9.130 __qdbl intrinsic on page 9-655.
9.131 __qsub intrinsic on page 9-656.
9.139 __sqrt intrinsic on page 9-665.
9.140 __sqrtf intrinsic on page 9-666.
9.141 __ssat intrinsic on page 9-667.
9.146 __usat intrinsic on page 9-674.
9.150 ARMv6 SIMD intrinsics on page 9-678.

3.9 Compiler support for European Telecommunications Standards Institute (ETSI) basic operations

ARM Compiler 4.1 and later provide support for the ETSI basic operations to help implement coding of speech.

ETSI has produced several recommendations for the coding of speech, for example, the G.723.1 and G.729 recommendations. These recommendations include source code and test sequences for reference implementations of the codecs.

Model implementations of speech codecs supplied by ETSI are based on a collection of C functions known as the *ETSI basic operations*. The ETSI basic operations include 16-bit, 32-bit and 40-bit operations for saturated arithmetic, 16-bit and 32-bit logical operations, and 16-bit and 32-bit operations for data type conversion.

Note

Version 2.0 of the ETSI collection of basic operations, as described in the *ITU-T Software Tool Library 2005 User's manual*, introduces new 16-bit, 32-bit and 40 bit-operations. These operations are not supported in the ARM compilation tools.

The ETSI basic operations serve as a set of primitives for developers publishing codec algorithms, rather than as a library for use by developers implementing codecs in C or C++.

ARM Compiler 4.1 and later provide support for the ETSI basic operations through the header file `dspfns.h`. The `dspfns.h` header file contains definitions of the ETSI basic operations as a combination of C code and intrinsics.

See `dspfns.h` for a complete list of the ETSI basic operations supported in ARM Compiler 4.1 and later.

ARM Compiler 4.1 and later support the original ETSI family of basic operations as described in the ETSI G.729 recommendation *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*, including:

- 16-bit and 32-bit saturated arithmetic operations, such as `add` and `sub`. For example, `add(v1, v2)` adds two 16-bit numbers `v1` and `v2` together, with overflow control and saturation, returning a 16-bit result.
- 16-bit and 32-bit multiplication operations, such as `mult` and `L_mult`. For example, `mult(v1, v2)` multiplies two 16-bit numbers `v1` and `v2` together, returning a scaled 16-bit result.
- 16-bit arithmetic shift operations, such as `shl` and `shr`. For example, the saturating left shift operation `shl(v1, v2)` arithmetically shifts the 16-bit input `v1` left `v2` positions. A negative shift count shifts `v1` right `v2` positions.
- 16-bit data conversion operations, such as `extract_l`, `extract_h`, and `round`. For example, `round(L_v1)` rounds the lower 16 bits of the 32-bit input `L_v1` into the most significant 16 bits with saturation.

  Note

  Beware that both the `dspfns.h` header file and the ISO C99 header file `math.h` both define (different versions of) the function `round()`. Take care to avoid this potential conflict.

Related concepts

3.1 Compiler intrinsics on page 3-64.
3.5 Compiler intrinsics for controlling IRQ and FIQ interrupts on page 3-68.
3.11 Texas Instruments (TI) C55x intrinsics for optimizing C code on page 3-75.
3.10 Overflow and carry status flags for C and C++ code on page 3-74.

Related references

3.2 Performance benefits of compiler intrinsics on page 3-65.

3.3 ARM assembler instruction intrinsics on page 3-66.

9.151 ETSI basic operations on page 9-679.

3.10 Overflow and carry status flags for C and C++ code

The implementation of the *European Telecommunications Standards Institute* (ETSI) basic operations in `dspfns.h` exposes the status flags `Overflow` and `Carry`.

These flags are available as global variables for use in your own C or C++ programs. For example:

```
#include <dspfns.h>          /* include ETSI intrinsics */
#include <stdio.h>
...
const int BUFLEN=255;
int a[BUFLEN], b[BUFLEN], c[BUFLEN];
...
Overflow = 0;                      /* clear overflow flag */
for (i = 0; i < BUFLEN; ++i) {
    c[i] = L_add(a[i], b[i]);      /* saturated add of a[i] and b[i] */
}
if (Overflow)
{
    fprintf(stderr, "Overflow on saturated addition\n");
}
```

Generally, saturating functions have a sticky effect on overflow. That is, the overflow flag remains set until it is explicitly cleared.

Related concepts

3.9 Compiler support for European Telecommunications Standards Institute (ETSI) basic operations on page 3-72.

3.11 Texas Instruments (TI) C55x intrinsics for optimizing C code

The ARM compilation tools support the emulation of selected TI C55x intrinsics.

The TI C55x compiler recognizes a number of intrinsics for the optimization of C code. The ARM compilation tools support the emulation of selected TI C55x intrinsics through the header file, `c55x.h`.

`c55x.h` gives a complete list of the TI C55x intrinsics that are emulated by the ARM compilation tools.

TI C55x intrinsics that are emulated in `c55x.h` include:

- Intrinsics for addition, subtraction, negation and absolute value, such as `_sadd` and `_ssub`. For example, `_sadd(v1, v2)` returns the 16-bit saturated sum of `v1` and `v2`.
- Intrinsics for multiplication and shifting, such as `_smpy` and `_sshl`. For example, `_smpy(v1, v2)` returns the saturated fractional-mode product of `v1` and `v2`.
- Intrinsics for rounding, saturation, bitcount and extremum, such as `_round` and `_count`. For example, `_round(v1)` returns the value `v1` rounded by adding 215 using unsaturated arithmetic, clearing the lower 16 bits.
- Associative variants of intrinsics for addition and multiply-and-accumulate. This includes all TI C55x intrinsics prefixed with `_a_`, for example, `_a_sadd` and `_a_smac`.
- Rounding variants of intrinsics for multiplication and shifting, for example, `_smacr` and `_smasr`.

The following TI C55x intrinsics are not supported in `c55x.h`:

- All **`long long`** variants of intrinsics. This includes all TI C55x intrinsics prefixed with `_ll`, for example, `_llsadd` and `_llshl`. **`long long`** variants of intrinsics are not supported in the ARM compilation tools because they operate on 40-bit data.
- All arithmetic intrinsics with side effects. For example, the TI C55x intrinsics `_firs` and `_lms` are not defined in `c55x.h`.
- Intrinsics for ETSI support functions, such as `L_add_c` and `L_sub_c`.

Note

An exception is the ETSI support function for saturating division, `divs`. This intrinsic is supported in `c55x.h`.

Related concepts

3.1 Compiler intrinsics on page 3-64.

3.5 Compiler intrinsics for controlling IRQ and FIQ interrupts on page 3-68.

3.9 Compiler support for European Telecommunications Standards Institute (ETSI) basic operations on page 3-72.

Related references

3.2 Performance benefits of compiler intrinsics on page 3-65.

3.3 ARM assembler instruction intrinsics on page 3-66.

Related information

Texas Instruments, `http://www.ti.com`.

3.12 Compiler support for accessing registers using named register variables

You can use named register variables to access registers of an ARM architecture-based processor.

Named register variables are declared by combining the **`register`** keyword with the `__asm` keyword. The `__asm` keyword takes one parameter, a character string, that names the register. For example, the following declaration declares `R0` as a named register variable for the register `r0`:

```
register int R0 __asm("r0");
```

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.

You must declare core registers as global rather than local named register variables. Your program might still compile if you declare them locally, but you risk unexpected runtime behavior if you do. There is no restriction on the scope of named register variables for other registers.

Note

A global named register variable is global to the source file in which it is declared, not global to the program. It has no effect on other files, unless you use multifile compilation or you declare it in a header file.

A typical use of named register variables is to access bits in the *Application Program Status Register* (APSR). The following example shows how to use named register variables to set the saturation flag `Q` in the APSR.

```
#ifndef __BIG_ENDIAN // bitfield layout of APSR is sensitive to endianness
typedef union
{
    struct
    {
        int mode:5;
        int T:1;
        int F:1;
        int I:1;
        int _dnm:19;
        int Q:1;
        int V:1;
        int C:1;
        int Z:1;
        int N:1;
    } b;
    unsigned int word;
} PSR;
#else /* __BIG_ENDIAN */
typedef union
{
    struct
    {
        int N:1;
        int Z:1;
        int C:1;
        int V:1;
        int Q:1;
        int _dnm:19;
        int I:1;
        int F:1;
        int T:1;
        int mode:5;
    } b;
    unsigned int word;
} PSR;
#endif /* __BIG_ENDIAN */
/* Declare PSR as a register variable for the "apsr" register */
register PSR apsr __asm("apsr");
void set_Q(void)
{
    apsr.b.Q = 1;
}
```

The following example shows how to use a named register variable to clear the Q flag in the APSR.

```
register unsigned int _apsr __asm("apsr");
void ClearQFlag(void)
{
    _apsr = _apsr & ~0x08000000; // clear Q flag
}
```

Compiling this example using `--cpu=7-M` results in the following assembly code:

```
ClearQFlag
    MRS    r0,APSR ; formerly CPSR
    BIC    r0,r0,#0x80000000
    MSR    APSR_nzcvq,r0; formerly CPSR_f
    BX     lr
```

The following example shows how to use named register variables to set up stack pointers.

```
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;        // set up Process Stack Pointer
}
```

Compiling this example using `--cpu=7-M` results in the following assembly code:

```
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
```

You can also use named register variables to access registers within a coprocessor. The string syntax within the declaration corresponds to how you intend to use the variable. For example, to declare a variable that you intend to use with the `MCR` instruction, look up the instruction syntax for this instruction and use this syntax when you declare your variable. The following example shows how to use a named register variable to set bits in a coprocessor register.

```
register unsigned int PMCR __asm("cp15:0:c9:c12:0");
void __reset_cycle_counter(void)
{
    PMCR = 4;
}
```

Compiling this example using `--cpu=7-M` results in the following assembly code:

```
__reset_cycle_counter PROC
    MOV    r0,#4
    MCR    p15,#0x0,r0,c9,c12,#0      ; move from r0 to c9
    BX     lr
    ENDP
```

In the above example, `PMCR` is declared as a register variable of type **`unsigned int`**, that is associated with the cp15 coprocessor, with `CRn = c9`, `CRm = c12`, `opcode1 = 0`, and `opcode2 = 0` in an `MCR` or `MRC` instruction. The `MCR` encoding in the disassembly corresponds with the register variable declaration.

The physical coprocessor register is specified with a combination of the two register numbers, `CRn` and `CRm`, and two opcode numbers. This maps to a single physical register.

The same principle applies if you want to manipulate individual bits in a register, but you write normal variable arithmetic in C, and the compiler does a read-modify-write of the coprocessor register. The following example shows how to manipulate bits in a coprocessor register using a named register variable

```
register unsigned int SCTLR __asm("cp15:0:c1:c0:0");
/* Set bit 11 of the system control register */
```

```
void enable_branch_prediction(void)
{
    SCTLR |= (1 << 11);
}
```

Compiling this example using `--cpu=7-M` results in the following assembly code:

```
__enable_branch_prediction PROC
    MRC    p15,#0x0,r0,c1,c0,#0
    ORR    r0,r0,#0x800
    MCR    p15,#0x0,r0,c1,c0,#0
    BX     lr
    ENDP
```

Related references

9.5 __asm on page 9-519.

9.156 Named register variables on page 9-685.

Related information

Application Program Status Register.

MRC and MRC2.

3.13 Pragmas recognized by the compiler

The compiler recognizes a number of pragmas, used to instruct the compiler to use particular features.

The compiler recognizes the following pragmas:

Pragmas for saving and restoring the pragma state

- `#pragma pop`
- `#pragma push`

Pragmas controlling optimization goals

- `#pragma Onum`
- `#pragma Ospace`
- `#pragma Otime`

Pragmas controlling code generation

- `#pragma arm`
- `#pragma thumb`
- `#pragma exceptions_unwind, #pragma no_exceptions_unwind`

Pragmas controlling loop unrolling

- `#pragma unroll [(n)]`
- `#pragma unroll_completely`

Pragmas controlling Precompiled Header (PCH) processing

- `#pragma hdrstop`
- `#pragma no_pch`

Pragmas controlling anonymous structures and unions

- `#pragma anon_unions, #pragma no_anon_unions`

Pragmas controlling diagnostic messages

- `#pragma diag_default tag[,tag,...]`
- `#pragma diag_error tag[,tag,...]`
- `#pragma diag_remark tag[,tag,...]`
- `#pragma diag_suppress tag[,tag,...]`
- `#pragma diag_warning tag[, tag, ...]`

Miscellaneous pragmas

- `#pragma arm section [section_type_list]`
- `#pragma import(__use_full_stdio)`
- `#pragma inline, #pragma no_inline`
- `#pragma once`
- `#pragma pack(n)`
- `#pragma softfp_linkage, #pragma no_softfp_linkage`
- `#pragma import symbol_name`

Related references

9.75 #pragma anon_unions, #pragma no_anon_unions on page 9-594.
9.76 #pragma arm on page 9-595.
9.77 #pragma arm section [section_type_list] on page 9-596.
9.78 #pragma diag_default tag[,tag,...] on page 9-598.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.

9.82 #pragma diag_warning tag[, tag, ...] on page 9-602.
9.83 #pragma exceptions_unwind, #pragma no_exceptions_unwind on page 9-603.
9.85 #pragma hdrstop on page 9-605.
9.86 #pragma import symbol_name on page 9-606.
9.87 #pragma import(__use_full_stdio) on page 9-607.
9.88 #pragma import(__use_smaller_memcpy) on page 9-608.
9.89 #pragma inline, #pragma no_inline on page 9-609.
9.90 #pragma no_pch on page 9-610.
9.91 #pragma Onum on page 9-611.
9.92 #pragma once on page 9-612.
9.93 #pragma Ospace on page 9-613.
9.94 #pragma Otime on page 9-614.
9.95 #pragma pack(n) on page 9-615.
9.96 #pragma pop on page 9-617.
9.97 #pragma push on page 9-618.
9.98 #pragma softfp_linkage, #pragma no_softfp_linkage on page 9-619.
9.99 #pragma thumb on page 9-620.
9.100 #pragma unroll [(n)] on page 9-621.
9.101 #pragma unroll_completely on page 9-623.
9.102 #pragma weak symbol, #pragma weak symbol1 = symbol2 on page 9-624.

3.14 Compiler and processor support for bit-banding

The compiler supports bit-banding for processors that provide the feature.

The compiler supports bit-banding in the following ways:

- `__attribute((bitband))` language extension.
- `--bitband` command-line option.

Bit-banding is a feature of the Cortex-M3 and Cortex-M4 processors (`--cpu=Cortex-M3` and `--cpu=Cortex-M4`) and some derivatives (for example, `--cpu=Cortex-M3-rev0`). This functionality is not available on other ARM processors.

Related concepts

3.15 Compiler type attribute, __attribute__((bitband)) on page 3-82.

3.16 --bitband compiler command-line option on page 3-83.

3.17 How the compiler handles bit-band objects placed outside bit-band regions on page 3-84.

Related references

9.56 __attribute__((bitband)) type attribute on page 9-575.

9.62 __attribute__((at(address))) variable attribute on page 9-581.

7.13 --bitband on page 7-283.

3.15 Compiler type attribute, __attribute__((bitband))

`__attribute__((bitband))` is a type attribute that lets you bit-band type definitions of structures.

In the following example, the unplaced bit-banded objects must be relocated into the bit-band region. This can be achieved by either using an appropriate scatter-loading description file or by using the `--rw_base` linker command-line option.

```
/* foo.c */
typedef struct {
  int i : 1;
  int j : 2;
  int k : 3;
} BB __attribute__((bitband));
BB value; // Unplaced object
void update_value(void)
{
  value.i = 1;
  value.j = 0;
}
/* end of foo.c */
```

Alternatively, you can use `__attribute__((at()))` to place bit-banded objects at a particular address in the bit-band region, as in the following example:

```
/* foo.c */
typedef struct {
  int i : 1;
  int j : 2;
  int k : 3;
} BB __attribute((bitband));
BB value __attribute__((at(0x20000040))); // Placed object
void update_value(void)
{
  value.i = 1;
  value.j = 0;
}
/* end of foo.c */
```

Related concepts

3.14 Compiler and processor support for bit-banding on page 3-81.

3.16 --bitband compiler command-line option on page 3-83.

3.17 How the compiler handles bit-band objects placed outside bit-band regions on page 3-84.

Related references

9.56 __attribute__((bitband)) type attribute on page 9-575.

9.62 __attribute__((at(address))) variable attribute on page 9-581.

7.13 --bitband on page 7-283.

Related information

Scatter-loading Features.

--rw_base=address linker option.

3.16 --bitband compiler command-line option

The `--bitband` command-line option bit-bands all non **`const`** global structure objects.

In the following example, when `--bitband` is applied to `foo.c`, the write to `value.i` is bit-banded. That is, the value `0x00000001` is written to the bit-band alias word that `value.i` maps to in the bit-band region.

Accesses to `value.j` and `value.k` are not bit-banded.

```
/* foo.c */
typedef struct {
  int i : 1;
  int j : 2;
  int k : 3;
} BB;
BB value __attribute__((at(0x20000040))); // Placed object
void update_value(void)
{
  value.i = 1;
  value.j = 0;
}
/* end of foo.c */
```

`armcc` supports the bit-banding of objects accessed through absolute addresses. When `--bitband` is applied to `foo.c` in the following example, the access to `rts` is bit-banded.

```
/* foo.c */
typedef struct {
  int rts : 1;
  int cts : 1;
  unsigned int data;
} uart;
#define com2 (*((volatile uart *)0x20002000))
void put_com2(int n)
{
  com2.rts = 1;
  com2.data = n;
}
/* end of foo.c */
```

Related concepts

3.14 Compiler and processor support for bit-banding on page 3-81.

3.15 Compiler type attribute, __attribute__((bitband)) on page 3-82.

3.17 How the compiler handles bit-band objects placed outside bit-band regions on page 3-84.

Related references

9.56 __attribute__((bitband)) type attribute on page 9-575.

9.62 __attribute__((at(address))) variable attribute on page 9-581.

7.13 --bitband on page 7-283.

3.17 How the compiler handles bit-band objects placed outside bit-band regions

Bit-band objects must not be placed outside bit-band regions.

If you do inadvertently place a bit-band object outside a bit-band region, either using the `at` attribute, or using an integer pointer to a particular address, the compiler responds as follows:

- If the `bitband` attribute is applied to an object type and `--bitband` is not specified on the command line, the compiler generates an error.
- If the `bitband` attribute is applied to an object type and `--bitband` is specified on the command line, the compiler generates a warning, and ignores the request to bit-band.
- If the `bitband` attribute is not applied to an object type and `--bitband` is specified on the command line, the compiler ignores the request to bit-band.

Related concepts

3.14 Compiler and processor support for bit-banding on page 3-81.

3.15 Compiler type attribute, __attribute__((bitband)) on page 3-82.

3.16 --bitband compiler command-line option on page 3-83.

Related references

9.56 __attribute__((bitband)) type attribute on page 9-575.

9.62 __attribute__((at(address))) variable attribute on page 9-581.

7.13 --bitband on page 7-283.

3.18 Compiler support for thread-local storage

Thread-local storage is a class of static storage that, like the stack, is private to each thread of execution.

Each thread in a process is given a location where it can store thread-specific data. Variables are allocated so that there is one instance of the variable for each existing thread.

Before each thread terminates, it releases its dynamic memory and any pointers to thread-local variables in that thread become invalid.

Related references

9.27 __declspec(thread) on page 9-544.

3.19 Compiler support for literal pools

Literal pools are areas of constant data in a code section.

No single instruction can generate a 4 byte constant, so the compiler generates code that loads these constants from a literal pool.

In the following example, the compiler generates code that loads the integer constant `0xdeadbeef` from a literal pool (marked with ***).

```
int f(void) {
  return 0xdeadbeef;
}


** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 12 bytes (alignment 4)
    Address: 0x00000000

    $a
    .text
    f
      0x00000000: e59f0000  .... LDR    r0,[pc,#0] ; [0x8] = 0xdeadbeef
      0x00000004: e12fff1e  ../. BX     lr
    $d
      0x00000008: deadbeef  .... DCD  3735928559            ***
```

An alternative to using literal pools is to generate the constant in a register with a `MOVW`/`MOVT` instruction pair:

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 12 bytes (alignment 4)
    Address: 0x00000000

    $a
    .text
    f
      0x00000000: e30b0eef  .... MOV    r0,#0xbeef
      0x00000004: e34d0ead  ..M. MOVT   r0,#0xdead
      0x00000008: e12fff1e  ../. BX     lr
```

In most cases, generating literal pools improves performance and code size. However, in some specific cases you might prefer to generate code without literal pools.

The following compiler options control literal pools:

- `--integer_literal_pools`.
- `--string_literal_pools`.
- `--branch_tables`.
- `--float_literal_pools`.

Related references

7.87 --integer_literal_pools, --no_integer_literal_pools on page 7-364.
7.158 --string_literal_pools, --no_string_literal_pools on page 7-441.
7.14 --branch_tables, --no_branch_tables on page 7-284.
7.63 --float_literal_pools, --no_float_literal_pools on page 7-337.

3.20 Compiler eight-byte alignment features

The compiler has the following eight-byte alignment features:

- The *Procedure Call Standard for the ARM Architecture* (AAPCS) requires that the stack is eight-byte aligned at all external interfaces. The compiler and C libraries preserve the eight-byte alignment of the stack. In addition, the default C library memory model maintains eight-byte alignment of the heap.
- Code is compiled in a way that requires and preserves the eight-byte alignment constraints at external interfaces.
- If you have assembly language files, or legacy objects, or libraries in your project, it is your responsibility to check that they preserve eight-byte stack alignment, and correct them if required.
- In RVCT v2.0 and later, and in ARM Compiler 4.1 and later, **`double`** and **`long long`** data types are eight-byte aligned for compliance with the *Application Binary Interface for the ARM Architecture* (AEABI). This enables efficient use of the `LDRD` and `STRD` instructions in ARMv5TE and later.
- The default implementations of `malloc()`, `realloc()`, and `calloc()` maintain an eight-byte aligned heap.
- The default implementation of `alloca()` returns an eight-byte aligned block of memory.

Related information

Procedure Call Standard for the ARM Architecture.

Application Binary Interface (ABI) for the ARM Architecture.

Alignment restrictions in load and store element and structure instructions.

alloca().

Section alignment with the linker.

3.21 Precompiled Header (PCH) files

Precompiled Header files can help reduce compilation time when the same header file is used by multiple source files.

Note

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

When you compile source files, the included header files are also compiled. If a header file is included in more than one source file, it is recompiled when each source file is compiled. Also, you might include header files that introduce many lines of code, but the primary source files that include them are relatively small. Therefore, it is often desirable to avoid recompiling a set of header files by precompiling them. These are referred to as PCH files.

The compiler can precompile and use PCH files automatically with the `--pch` option, or you can use the `--create_pch` and `--use_pch` options to manually control the use of PCH files.

By default, when the compiler creates a PCH file, it:
- Takes the name of the primary source file and replaces the suffix with `.pch`.
- Creates the file in the same directory as the primary source file.

Note

Support for PCH processing is not available when you specify multiple source files in a single compilation. In such a case, the compiler issues an error message and aborts the compilation.

Note

Do not assume that if a PCH file is available, it is used by the compiler. In some cases, system configuration issues mean that the compiler might not always be able to use the PCH file. Address Space Randomization on *Red Hat Enterprise Linux 3* (RHE3) is one example of a possible system configuration issue.

Related concepts

2.4 Order of compiler command-line options on page 2-43.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.

3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.

Related references

7.131 --pch_messages, --no_pch_messages on page 7-413.
9.85 #pragma hdrstop on page 9-605.
9.90 #pragma no_pch on page 9-610.
7.129 --pch on page 7-411.
7.130 --pch_dir=dir on page 7-412.
7.132 --pch_verbose, --no_pch_verbose on page 7-414.

3.22 Automatic Precompiled Header (PCH) file processing

The `--pch` command-line option enables automatic PCH file processing.

——— **Note** ———

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

———————

Automatic PCH file processing means that the compiler automatically looks for a qualifying PCH file, and reads it if found. Otherwise, the compiler creates one for use on a subsequent compilation.

When the compiler creates a PCH file, it takes the name of the primary source file and replaces the suffix with `.pch`. The PCH file is created in the directory of the primary source file unless the `--pch_dir` option is specified.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.21 Precompiled Header (PCH) files on page 3-88.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.

Related references

7.129 --pch on page 7-411.
7.130 --pch_dir=dir on page 7-412.

3.23 Precompiled Header (PCH) file processing and the header stop point

The PCH file contains a snapshot of all the code that precedes a *header stop point*.

Note

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

Typically, the header stop point is the first token in the primary source file that does not belong to a preprocessing directive. In the following example, the header stop point is `int` and the PCH file contains a snapshot that reflects the inclusion of `xxx.h` and `yyy.h`:

```
#include "xxx.h"
#include "yyy.h"
int i;
```

You can manually specify the header stop point with `#pragma hdrstop`. If you use this pragma, it must appear before the first token that does not belong to a preprocessing directive. In this example, it must be placed before `int`, as follows:

```
#include "xxx.h"
#include "yyy.h"
#pragma hdrstop
int i;
```

If a conditional directive block (`#if`, `#ifdef`, or `#ifndef`) encloses the first non-preprocessor token or `#pragma hdrstop`, the header stop point is the outermost enclosing conditional directive.

For example:

```
#include "xxx.h"
#ifndef YYY_H
#define YYY_H 1
#include "yyy.h"
#endif
#if TEST /* Header stop point lies immediately before #if TEST */
int i;
#endif
```

In this example, the first token that does not belong to a preprocessing directive is `int`, but the header stop point is the start of the `#if` block containing it. The PCH file reflects the inclusion of `xxx.h` and, conditionally, the definition of `YYY_H` and inclusion of `yyy.h`. It does not contain the state produced by `#if TEST`.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.

3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.

Related references
7.129 --pch on page 7-411.
7.130 --pch_dir=dir on page 7-412.
9.85 #pragma hdrstop on page 9-605.

3.24 Precompiled Header (PCH) file creation requirements

A PCH file is produced only if the header stop point and the code preceding it, mainly the header files, meet specific requirements.

Note

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

These requirements are as follows:

- The header stop point must appear at file scope. It must not be within an unclosed scope established by a header file. For example, a PCH file is not created in this case:

```
// xxx.h
class A
{
    // xxx.c
    #include "xxx.h"
    int i;
};
```

- The header stop point must not be inside a declaration that is started within a header file. Also, in C++, it must not be part of a declaration list of a linkage specification. For example, in the following case the header stop point is `int`, but because it is not the start of a new declaration, no PCH file is created:

```
// yyy.h
static
// yyy.c
#include "yyy.h"
int i;
```

- The header stop point must not be inside a `#if` block or a `#define` that is started within a header file.
- The processing that precedes the header stop point must not have produced any errors.

  Note

  Warnings and other diagnostics are not reproduced when the PCH file is reused.

- No references to predefined macros `__DATE__` or `__TIME__` must appear.
- No instances of the `#line` preprocessing directive must appear.
- `#pragma no_pch` must not appear.
- The code preceding the header stop point must have introduced a sufficient number of declarations to justify the overhead associated with precompiled headers.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.

3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.

Related references

7.129 --pch on page 7-411.
7.130 --pch_dir=dir on page 7-412.
7.131 --pch_messages, --no_pch_messages on page 7-413.
7.132 --pch_verbose, --no_pch_verbose on page 7-414.
9.85 #pragma hdrstop on page 9-605.
9.90 #pragma no_pch on page 9-610.

3.25 Compilation with multiple Precompiled Header (PCH) files

More than one PCH file might apply to a given compilation. If so, the compiler uses the largest PCH file.

Note

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

That is, the compiler uses the PCH file representing the most preprocessing directives from the primary source file.

For example, a primary source file might begin with:

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

If there is one PCH file for `xxx.h` and a second for `xxx.h` and `yyy.h`, the latter PCH file is selected, assuming that both apply to the current compilation. Additionally, after the PCH file for the first two headers is read in and the third is compiled, a new PCH file for all three headers is created if the requirements for PCH file creation are met.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.

3.26 Obsolete Precompiled Header (PCH) files

In automatic PCH processing mode the compiler identifies and deletes obsolete PCH files.

——— **Note** ———

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

————————

The compiler indicates that a PCH file is obsolete, and deletes it, under the following circumstances:

- If the PCH file is based on at least one out-of-date header file but is otherwise applicable for the current compilation.
- If the PCH file has the same base name as the source file being compiled, for example, `xxx.pch` and `xxx.c`, but is not applicable for the current compilation, for example, because you have used different command-line options.

These describe some common cases. You must delete other PCH files as required.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.

3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file

You can manually specify the filename and location of PCH files for the compiler to create and use.

——— **Note** ———

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

———————————

Use the following compiler command-line options to specify PCH filenames and locations:

- `--create_pch=`*`filename`*
- `--pch_dir=`*`directory`*
- `--use_pch=`*`filename`*

If you use `--create_pch` or `--use_pch` with the `--pch_dir` option, the indicated filename is appended to the directory name, unless the filename is an absolute path name.

——— **Note** ———

If multiple options are specified on the same command line, the following rules apply:

- `--use_pch` takes precedence over `--pch`.
- `--create_pch` takes precedence over all other PCH file options.

———————————

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.

Related references

7.30 --create_pch=filename on page 7-304.
7.129 --pch on page 7-411.
7.130 --pch_dir=dir on page 7-412.
7.166 --use_pch=filename on page 7-451.

3.28 Selectively applying Precompiled Header (PCH) file processing

You can selectively include and exclude header files for PCH file processing, even if you are using automatic PCH file processing.

Note

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

Use the `#pragma hdrstop` directive to insert a manual header stop point in the primary source file. Insert it before the first token that does not belong to a preprocessing directive. This enables you to specify where the set of header files that is subject to precompilation ends. For example,

```
#include "xxx.h"
#include "yyy.h"
#pragma hdrstop
#include "zzz.h"
```

In this example, the PCH file includes the processing state for `xxx.h` and `yyy.h` but not for `zzz.h`. This is useful if you decide that the information following the `#pragma hdrstop` does not justify the creation of another PCH file.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.

Related references

9.85 #pragma hdrstop on page 9-605.
9.90 #pragma no_pch on page 9-610.

3.29 Suppressing Precompiled Header (PCH) file processing

To suppress PCH file processing, use the `#pragma no_pch` directive in the primary source file.

Note

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

You do not have to place this directive at the beginning of the file for it to take effect. For example, no PCH file is created if you compile the following source code with `armcc --create_pch=foo.pch myprog.c`:

```
#include "xxx.h"
#pragma no_pch
#include "zzz.h"
```

If you want to selectively enable PCH processing, for example, subject `xxx.h` to PCH file processing, but not `zzz.h`, replace `#pragma no_pch` with `#pragma hdrstop`, as follows:

```
#include "xxx.h"
#pragma hdrstop
#include "zzz.h"
```

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.

Related references

9.85 #pragma hdrstop on page 9-605.
9.90 #pragma no_pch on page 9-610.

3.30 Message output during Precompiled Header (PCH) processing

Whenever the compiler creates or uses a PCH file, it displays a message. You can suppress these messages or make them more verbose.

Note

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

When the compiler creates or uses a PCH file, it displays the following kind of message:

```
test.c: creating precompiled header file test.pch
```

You can suppress this message with the compiler command-line option `--no_pch_messages`.

The `--pch_verbose` option enables verbose mode. In verbose mode, the compiler displays a message for each PCH file that it considers but does not use, giving the reason why it cannot be used.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.
3.31 Performance issues with Precompiled Header (PCH) files on page 3-101.

Related references

7.131 --pch_messages, --no_pch_messages on page 7-413.
7.132 --pch_verbose, --no_pch_verbose on page 7-414.

3.31 Performance issues with Precompiled Header (PCH) files

Typically, the overhead of creating and reading a PCH file is small, even for reasonably large header files. If the PCH file is used, there is typically a significant decrease in compilation time. However, PCH files can range in size from about 250KB to several megabytes or more, so you might not want to create many PCH files.

Note

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

PCH processing might not always be appropriate, for example, where you have an arbitrary set of files with non-uniform initial sequences of preprocessing directives.

The benefits of PCH processing occur when several source files can share the same PCH file. The more sharing, the less disk space is consumed. Sharing minimizes the disadvantage of large PCH files, without giving up the advantage of a significant decrease in compilation times.

Therefore, to take full advantage of header file precompilation, you might have to re-order the `#include` sections of your source files, or group `#include` directives within a commonly used header file.

Different environments and different projects might have differing requirements. Be aware, however, that making the best use of PCH support might require some experimentation and probably some minor changes to source code.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.
3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.25 Compilation with multiple Precompiled Header (PCH) files on page 3-95.
3.26 Obsolete Precompiled Header (PCH) files on page 3-96.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.

3.32 Default compiler options that are affected by optimization level

In general, optimization levels are independent from the default behavior of command-line options. However, there are a small number of exceptions where the level of optimization you use changes the default option.

These exceptions are:

- `--autoinline`, `--no_autoinline`.
- `--data_reorder`, `--no_data_reorder`.

Depending on the value of `-O`*`num`* you use (`-O0`, `-O1`, `-O2`, or `-O3`), the default option changes as specified. See the individual command-line option reference descriptions for more information.

Related references

7.11 --autoinline, --no_autoinline on page 7-281.

7.32 --data_reorder, --no_data_reorder on page 7-306.

7.119 -Onum on page 7-399.

Chapter 4
Compiler Coding Practices

Describes programming techniques and practices to help you increase the portability, efficiency and robustness of your C and C++ source code.

It contains the following sections:

- *4.1 The compiler as an optimizing compiler* on page 4-106.
- *4.2 Compiler optimization for code size versus speed* on page 4-107.
- *4.3 Compiler optimization levels and the debug view* on page 4-108.
- *4.4 Selecting the target processor at compile time* on page 4-111.
- *4.5 Enabling FPU for bare-metal* on page 4-112.
- *4.6 Optimization of loop termination in C code* on page 4-113.
- *4.7 Loop unrolling in C code* on page 4-115.
- *4.8 Compiler optimization and the volatile keyword* on page 4-117.
- *4.9 Code metrics* on page 4-119.
- *4.10 Code metrics for measurement of code size and data size* on page 4-120.
- *4.11 Stack use in C and C++* on page 4-121.
- *4.12 Benefits of reducing debug information in objects and libraries* on page 4-123.
- *4.13 Methods of reducing debug information in objects and libraries* on page 4-124.
- *4.14 Guarding against multiple inclusion of header files* on page 4-125.
- *4.15 Methods of minimizing function parameter passing overhead* on page 4-126.
- *4.16 Returning structures from functions through registers* on page 4-127.
- *4.17 Functions that return the same result when called with the same arguments* on page 4-128.
- *4.18 Comparison of pure and impure functions* on page 4-129.
- *4.19 Recommendation of postfix syntax when qualifying functions with ARM function modifiers* on page 4-130.
- *4.20 Inline functions* on page 4-131.
- *4.21 Compiler decisions on function inlining* on page 4-132.

- *4.22 Automatic function inlining and static functions* on page 4-133.
- *4.23 Inline functions and removal of unused out-of-line functions at link time* on page 4-134.
- *4.24 Automatic function inlining and multifile compilation* on page 4-135.
- *4.25 Restriction on overriding compiler decisions about function inlining* on page 4-136.
- *4.26 Compiler modes and inline functions* on page 4-137.
- *4.27 Inline functions in C++ and C90 mode* on page 4-138.
- *4.28 Inline functions in C99 mode* on page 4-139.
- *4.29 Inline functions and debugging* on page 4-141.
- *4.30 Types of data alignment* on page 4-142.
- *4.31 Advantages of natural data alignment* on page 4-143.
- *4.32 Compiler storage of data objects by natural byte alignment* on page 4-144.
- *4.33 Relevance of natural data alignment at compile time* on page 4-145.
- *4.34 Unaligned data access in C and C++ code* on page 4-146.
- *4.35 The __packed qualifier and unaligned data access in C and C++ code* on page 4-147.
- *4.36 Unaligned fields in structures* on page 4-148.
- *4.37 Performance penalty associated with marking whole structures as packed* on page 4-149.
- *4.38 Unaligned pointers in C and C++ code* on page 4-150.
- *4.39 Unaligned Load Register (LDR) instructions generated by the compiler* on page 4-151.
- *4.40 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 4-152.
- *4.41 Compiler support for floating-point arithmetic* on page 4-154.
- *4.42 Default selection of hardware or software floating-point support* on page 4-156.
- *4.43 Example of hardware and software support differences for floating-point arithmetic* on page 4-157.
- *4.44 Vector Floating-Point (VFP) architectures* on page 4-159.
- *4.45 Limitations on hardware handling of floating-point arithmetic* on page 4-160.
- *4.46 Implementation of Vector Floating-Point (VFP) support code* on page 4-161.
- *4.47 Compiler and library support for half-precision floating-point numbers* on page 4-163.
- *4.48 Half-precision floating-point number format* on page 4-164.
- *4.49 Compiler support for floating-point computations and linkage* on page 4-165.
- *4.50 Types of floating-point linkage* on page 4-166.
- *4.51 Compiler options for floating-point linkage and computations* on page 4-167.
- *4.52 Floating-point linkage and computational requirements of compiler options* on page 4-169.
- *4.53 Processors and their implicit Floating-Point Units (FPUs)* on page 4-171.
- *4.54 Integer division-by-zero errors in C code* on page 4-173.
- *4.55 Software floating-point division-by-zero errors in C code* on page 4-175.
- *4.56 About trapping software floating-point division-by-zero errors* on page 4-176.
- *4.57 Identification of software floating-point division-by-zero errors* on page 4-177.
- *4.58 Software floating-point division-by-zero debugging* on page 4-179.
- *4.59 New language features of C99* on page 4-180.
- *4.60 New library features of C99* on page 4-182.
- *4.61 // comments in C99 and C90* on page 4-183.
- *4.62 Compound literals in C99* on page 4-184.
- *4.63 Designated initializers in C99* on page 4-185.
- *4.64 Hexadecimal floating-point numbers in C99* on page 4-186.
- *4.65 Flexible array members in C99* on page 4-187.
- *4.66 __func__ predefined identifier in C99* on page 4-188.
- *4.67 inline functions in C99* on page 4-189.
- *4.68 long long data type in C99 and C90* on page 4-190.
- *4.69 Macros with a variable number of arguments in C99* on page 4-191.
- *4.70 Mixed declarations and statements in C99* on page 4-192.
- *4.71 New block scopes for selection and iteration statements in C99* on page 4-193.
- *4.72 _Pragma preprocessing operator in C99* on page 4-194.
- *4.73 Restricted pointers in C99* on page 4-195.
- *4.74 Additional <math.h> library functions in C99* on page 4-196.
- *4.75 Complex numbers in C99* on page 4-197.

- *4.76 Boolean type and <stdbool.h> in C99* on page 4-198.
- *4.77 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99* on page 4-199.
- *4.78 <fenv.h> floating-point environment access in C99* on page 4-200.
- *4.79 <stdio.h> snprintf family of functions in C99* on page 4-201.
- *4.80 <tgmath.h> type-generic math macros in C99* on page 4-202.
- *4.81 <wchar.h> wide character I/O functions in C99* on page 4-203.
- *4.82 How to prevent uninitialized data from being initialized to zero* on page 4-204.

4.1 The compiler as an optimizing compiler

The compiler is highly optimizing for small code size and high performance, performing a range of optimization techniques.

The compiler performs optimizations common to other optimizing compilers, for example, data-flow optimizations such as common sub-expression elimination and loop optimizations such as loop combining and distribution.

In addition, the compiler performs a range of optimizations specific to ARM architecture-based processors.

Although the compiler performs a number of architecture independent optimizations, you can often significantly improve the performance of your C or C++ code by selecting correct optimization criteria, and the correct target processor and architecture.

Note

Optimization options can limit debug information generated by the compiler.

Related concepts

4.2 Compiler optimization for code size versus speed on page 4-107.
4.3 Compiler optimization levels and the debug view on page 4-108.
4.6 Optimization of loop termination in C code on page 4-113.
4.8 Compiler optimization and the volatile keyword on page 4-117.

Related tasks

4.4 Selecting the target processor at compile time on page 4-111.

Related references

7.11 --autoinline, --no_autoinline on page 7-281.
7.29 --cpu=name compiler option on page 7-302.
7.32 --data_reorder, --no_data_reorder on page 7-306.
7.65 --forceinline on page 7-339.
7.67 --fpmode=model on page 7-341.
7.86 --inline, --no_inline on page 7-363.
7.93 --library_interface=lib on page 7-370.
7.94 --library_type=lib on page 7-372.
7.106 --lower_ropi, --no_lower_ropi on page 7-385.
7.107 --lower_rwpi, --no_lower_rwpi on page 7-386.
7.114 --multifile, --no_multifile on page 7-393.
7.119 -Onum on page 7-399.
7.124 -Ospace on page 7-406.
7.125 -Otime on page 7-407.
7.146 --retain=option on page 7-428.

4.2 Compiler optimization for code size versus speed

The compiler can optimize for either code size or performance.

The following options control whether the compiler optimizes for code size or performance:

`-Ospace`
: This option causes the compiler to optimize mainly for code size. This is the default option.

`-Otime`
: This option causes the compiler to optimize mainly for speed.

For best results, you must build your application using the most appropriate command-line option.

Note

These command-line options instruct the compiler to use optimizations that deliver the effect wanted in the vast majority of cases. However, it is not guaranteed that `-Otime` always generates faster code, or that `-Ospace` always generates smaller code.

Related references

7.124 -Ospace on page 7-406.

7.125 -Otime on page 7-407.

4.3 Compiler optimization levels and the debug view

The precise optimizations performed by the compiler depend both on the level of optimization chosen, and whether you are optimizing for performance or code size.

The compiler supports the following optimization levels:

`0`

Minimum optimization. Turns off most optimizations. When debugging is enabled, this option gives the best possible debug view because the structure of the generated code directly corresponds to the source code. All optimization that interferes with the debug view is disabled. In particular:

- Breakpoints can be set on any reachable point, including dead code.
- The value of a variable is available everywhere within its scope, except where it is uninitialized.
- Backtrace gives the stack of open function activations that is expected from reading the source.

——— **Note** ———

Although the debug view produced by `-O0` corresponds most closely to the source code, users might prefer the debug view produced by `-O1` because this improves the quality of the code without changing the fundamental structure.

———————

——— **Note** ———

Dead code includes reachable code that has no effect on the result of the program, for example an assignment to a local variable that is never used. Unreachable code is specifically code that cannot be reached via any control flow path, for example code that immediately follows a return statement.

———————

`1`

Restricted optimization. The compiler only performs optimizations that can be described by debug information. Removes unused inline functions and unused static functions. Turns off optimizations that seriously degrade the debug view. If used with `--debug`, this option gives a generally satisfactory debug view with good code density.
The differences in the debug view from `–O0` are:

- Breakpoints cannot be set on dead code.
- Values of variables might not be available within their scope after they have been initialized. For example if their assigned location has been reused.
- Functions with no side-effects might be called out of sequence, or might be omitted if the result is not needed.
- Backtrace might not give the stack of open function activations that is expected from reading the source because of the presence of tailcalls.

The optimization level `–O1` produces good correspondence between source code and object code, especially when the source code contains no dead code. The generated code can be significantly smaller than the code at `–O0`, which can simplify analysis of the object code.

2

High optimization. If used with `--debug`, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler might perform optimizations that cannot be described by debug information.

This is the default optimization level.

The differences in the debug view from `-O1` are:

- The source code to object code mapping might be many to one, because of the possibility of multiple source code locations mapping to one point of the file, and more aggressive instruction scheduling.
- Instruction scheduling is allowed to cross sequence points. This can lead to mismatches between the reported value of a variable at a particular point, and the value you might expect from reading the source code.
- The compiler automatically inlines functions.

3

Maximum optimization. When debugging is enabled, this option typically gives a poor debug view. ARM recommends debugging at lower optimization levels.
If you use `-O3` and `-Otime` together, the compiler performs extra optimizations that are more aggressive, such as:

- High-level scalar optimizations, including loop unrolling. 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.

These optimizations effectively rewrite the input source code, resulting in object code with the lowest correspondence to source code and the worst debug view. The `--loop_optimization_level=option` controls the amount of loop optimization performed at `-O3 -Otime`. The higher the amount of loop optimization the worse the correspondence between source and object code.

Use of the `--vectorize` option also lowers the correspondence between source and object code.

For extra information about the high level transformations performed on the source code at `-O3 -Otime` use the `--remarks` command-line option.

Because optimization affects the mapping of object code to source code, the choice of optimization level with `-Ospace` and `-Otime` generally impacts the debug view.

The option `-O0` is the best option to use if a simple debug view is required. Selecting `-O0` typically increases the size of the ELF image by 7 to 15%. To reduce the size of your debug tables, use the `--remove_unneeded_entities` option.

Related concepts

4.12 Benefits of reducing debug information in objects and libraries on page 4-123.

Related references

4.13 Methods of reducing debug information in objects and libraries on page 4-124.
7.33 --debug, --no_debug on page 7-307.
7.34 --debug_macros, --no_debug_macros on page 7-308.
7.51 --dwarf2 on page 7-325.
7.52 --dwarf3 on page 7-326.
7.119 -Onum on page 7-399.
7.124 -Ospace on page 7-406.
7.125 -Otime on page 7-407.
7.144 --remove_unneeded_entities, --no_remove_unneeded_entities on page 7-426.

Related information

ELF for the ARM Architecture.

4.4 Selecting the target processor at compile time

You can often significantly improve the performance of your C or C++ code by selecting the appropriate target processor at compile time.

Each new version of the ARM architecture typically supports extra instructions, extra modes of operation, pipeline differences, and register renaming.

Procedure

1. Decide whether the compiled program is to run on a specific ARM architecture-based processor or on different ARM processors.
2. Obtain the name, or names, of the target processors recognized by the compiler using the following compiler command-line option:

   ```
   --cpu=list
   ```
3. If the compiled program is to run on a specific ARM architecture-based processor, having obtained the name of the processor with the `--cpu=list` option, select the target processor using the `--cpu=`*`name`* compiler command-line option.

   For example, to compile code to run on a Cortex-A9 processor:

   ```
   armcc --cpu=Cortex-A9 myprog.c
   ```

   Alternatively, if the compiled program is to run on different ARM processors, choose the lowest common denominator architecture appropriate for the application and then specify that architecture in place of the processor name. For example, to compile code for processors supporting the ARMv6 architecture:

   ```
   armcc --cpu=6 myprog.c
   ```

Selecting the target processor using the `--cpu=`*`name`* command-line option lets the compiler:
- Make full use of all available instructions for that particular processor.
- Perform processor-specific optimizations such as instruction scheduling.

`--cpu=list` lists all the processors and architectures that the compiler supports.

Related references

7.28 --cpu=list on page 7-301.

7.29 --cpu=name compiler option on page 7-302.

4.5 Enabling FPU for bare-metal

If the compiler knows that an FPU is available, for example if you use the `--cpu` option to specify a processor with an FPU, then the compiler might introduce FPU instructions into your code.

These instructions can be introduced even if you are not deliberately performing any floating-point operations.

If you want to build an image that does not use any FPU instructions, and does not require that the FPU be enabled, you can use the `--fpu=none` option when building all your source files.

When targeting bare-metal and compiling for a processor with an FPU, you must enable the FPU in your startup code before you can execute FPU instructions. See the *Technical Reference Manual* for your processor.

For example, the following startup code enables FPU hardware for a Cortex-M3 processor (TBD: Need code for M3):

```
__asm void StartHere(void)
{
MRC p15,0,r0,c1,c0,2    // Read CP Access register
ORR r0,r0,#0x00f00000   // Enable full access to NEON/VFP (Coprocessors 10 and 11)
MCR p15,0,r0,c1,c0,2    // Write CP Access register
ISB
MOV r0,#0x40000000      // Switch on the VFP and NEON hardware
MSR FPEXC,r0            // Set EN bit in FPEXC
IMPORT __main
B __main                // Enter normal C run-time environment & library start-up
}
```

To compile this code:

```
armcc -c --cpu=Cortex-M3 main.c
armlink --entry=StartHere main.o
```

Related tasks

4.4 Selecting the target processor at compile time on page 4-111.

Related references

7.29 --cpu=name compiler option on page 7-302.

7.69 --fpu=name compiler option on page 7-344.

Related information

--startup=symbol, --no_startup linker option.

4.6 Optimization of loop termination in C code

Loops are a common construct in most programs. Because a significant amount of execution time is often spent in loops, it is worthwhile paying attention to time-critical loops.

The loop termination condition can cause significant overhead if written without caution. Where possible:

- Use simple termination conditions.
- Write count-down-to-zero loops.
- Use counters of type **`unsigned int`**.
- Test for equality against zero.

Following any or all of these guidelines, separately or in combination, is likely to result in better code.

The following table shows two sample implementations of a routine to calculate `n!` that together illustrate loop termination overhead. The first implementation calculates `n!` using an incrementing loop, while the second routine calculates `n!` using a decrementing loop.

Table 4-1 C code for incrementing and decrementing loops

Incrementing loop

```
int fact1(int n)
{
    int i, fact = 1;
    for (i = 1; i <= n; i++)
        fact *= i;
    return (fact);
}
```

Decrementing loop

```
int fact2(int n)
{
    unsigned int i, fact = 1;
    for (i = n; i != 0; i--)
        fact *= i;
    return (fact);
}
```

The following table shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations above, where the C code for both implementations has been compiled using the options `-O2 -Otime`.

Table 4-2 C Disassembly for incrementing and decrementing loops

Incrementing loop

```
fact1 PROC
    MOV      r2, r0
    MOV      r0, #1
    CMP      r2, #1
    MOV      r1, r0
    BXLT     lr
|L1.20|
    MUL      r0, r1, r0
    ADD      r1, r1, #1
    CMP      r1, r2
    BLE      |L1.20|
    BX       lr
    ENDP
```

Decrementing loop

```
fact2 PROC
    MOVS     r1, r0
    MOV      r0, #1
    BXEQ     lr
|L1.12|
    MUL      r0, r1, r0
    SUBS     r1, r1, #1
    BNE      |L1.12|
    BX       lr
    ENDP
```

Comparing the disassemblies shows that the `ADD` and `CMP` instruction pair in the incrementing loop disassembly has been replaced with a single `SUBS` instruction in the decrementing loop disassembly. This is because a compare with zero can be used instead.

In addition to saving an instruction in the loop, the variable `n` does not have to be saved across the loop, so the use of a register is also saved in the decrementing loop disassembly. This eases register allocation. It is even more important if the original termination condition involves a function call. For example:

```
for (...; i < get_limit(); ...);
```

The technique of initializing the loop counter to the number of iterations required, and then decrementing down to zero, also applies to **`while`** and **`do`** statements.

Related concepts

4.7 Loop unrolling in C code on page 4-115.

4.7 Loop unrolling in C code

Loops are a common construct in most programs. Because a significant amount of execution time is often spent in loops, it is worthwhile paying attention to time-critical loops.

Small loops can be unrolled for higher performance, with the disadvantage of increased code size. When a loop is unrolled, a loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled so that the loop overhead completely disappears. The compiler unrolls loops automatically at `-O3 -Otime`. Otherwise, any unrolling must be done in source code.

Note

Manual unrolling of loops might hinder the automatic re-rolling of loops and other loop optimizations by the compiler.

The advantages and disadvantages of loop unrolling can be illustrated using the two sample routines shown in the following table. Both routines efficiently test a single bit by extracting the lowest bit and counting it, after which the bit is shifted out.

The first implementation uses a loop to count bits. The second routine is the first implementation unrolled four times, with an optimization applied by combining the four shifts of `n` into one shift.

Unrolling frequently provides new opportunities for optimization.

Table 4-3 C code for rolled and unrolled bit-counting loops

Bit-counting loop	Unrolled bit-counting loop

```
int countbit1(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        n >>= 1;
    }
    return bits;
}
```
	```
	int countbit2(unsigned int n)
	{
	    int bits = 0;
	    while (n != 0)
	    {
	        if (n & 1) bits++;
	        if (n & 2) bits++;
	        if (n & 4) bits++;
	        if (n & 8) bits++;
	        n >>= 4;
	    }
	    return bits;
	}
	```

The following table shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations above, where the C code for each implementation has been compiled using the option `-O2`.

Table 4-4 Disassembly for rolled and unrolled bit-counting loops

Bit-counting loop

```
countbit1 PROC
    MOV      r1, #0
    B        |L1.20|
|L1.8|
    TST      r0, #1
    ADDNE    r1, r1, #1
    LSR      r0, r0, #1
|L1.20|
    CMP      r0, #0
    BNE      |L1.8|
    MOV      r0, r1
    BX       lr
    ENDP
```

Unrolled bit-counting loop

```
countbit2 PROC
    MOV      r1, r0
    MOV      r0, #0
    B        |L1.48|
|L1.12|
    TST      r1, #1
    ADDNE    r0, r0, #1
    TST      r1, #2
    ADDNE    r0, r0, #1
    TST      r1, #4
    ADDNE    r0, r0, #1
    TST      r1, #8
    ADDNE    r0, r0, #1
    LSR      r1, r1, #4
|L1.48|
    CMP      r1, #0
    BNE      |L1.12|
    BX       lr
    ENDP
```

On the ARM9 processor, checking a single bit takes six cycles in the disassembly of the bit-counting loop shown in the leftmost column. The code size is only nine instructions. The unrolled version of the bit-counting loop checks four bits at a time per loop iteration, taking on average only three cycles per bit. However, the cost is the larger code size of fifteen instructions.

Related concepts

4.6 Optimization of loop termination in C code on page 4-113.

4.8 Compiler optimization and the volatile keyword

Higher optimization levels can reveal problems in some programs that are not apparent at lower optimization levels, for example, missing **`volatile`** qualifiers.

This can manifest itself in a number of ways. Code might become stuck in a loop while polling hardware, multi-threaded code might exhibit strange behavior, or optimization might result in the removal of code that implements deliberate timing delays. In such cases, it is possible that some variables are required to be declared as **`volatile`**.

The declaration of a variable as **`volatile`** tells the compiler that the variable can be modified at any time externally to the implementation, for example, by the operating system, by another thread of execution such as an interrupt routine or signal handler, or by hardware. Because the value of a **`volatile`**-qualified variable can change at any time, the actual variable in memory must always be accessed whenever the variable is referenced in code. This means the compiler cannot perform optimizations on the variable, for example, caching its value in a register to avoid memory accesses. Similarly, when used in the context of implementing a sleep or timer delay, declaring a variable as **`volatile`** tells the compiler that a specific type of behavior is intended, and that such code must not be optimized in such a way that it removes the intended functionality.

In contrast, when a variable is not declared as **`volatile`**, the compiler can assume its value cannot be modified in unexpected ways. Therefore, the compiler can perform optimizations on the variable.

The use of the **`volatile`** keyword is illustrated in the two sample routines of the following table. Both of these routines loop reading a buffer until a status flag `buffer_full` is set to true. The state of `buffer_full` can change asynchronously with program flow.

The two versions of the routine differ only in the way that `buffer_full` is declared. The first routine version is incorrect. Notice that the variable `buffer_full` is not qualified as **`volatile`** in this version. In contrast, the second version of the routine shows the same loop where `buffer_full` is correctly qualified as **`volatile`**.

Table 4-5 C code for nonvolatile and volatile buffer loops

Nonvolatile version of buffer loop

```
int buffer_full;
int read_stream(void)
{
    int count = 0;
    while (!buffer_full)
    {
        count++;
    }
    return count;
}
```

Volatile version of buffer loop

```
volatile int buffer_full;
int read_stream(void)
{
    int count = 0;
    while (!buffer_full)
    {
        count++;
    }
    return count;
}
```

The following table shows the corresponding disassembly of the machine code produced by the compiler for each of the examples above, where the C code for each implementation has been compiled using the option `-O2`.

Table 4-6 Disassembly for nonvolatile and volatile buffer loop

Nonvolatile version of buffer loop

```
read_stream PROC
    LDR      r1, |L1.28|
    MOV      r0, #0
    LDR      r1, [r1, #0]
|L1.12|
    CMP      r1, #0
    ADDEQ    r0, r0, #1
    BEQ      |L1.12|      ; infinite loop
    BX       lr
    ENDP
|L1.28|
    DCD      ||.data||
    AREA ||.data||, DATA, ALIGN=2
buffer_full
    DCD      0x00000000
```

Volatile version of buffer loop

```
read_stream PROC
    LDR      r1, |L1.28|
    MOV      r0, #0
|L1.8|
    LDR      r2, [r1, #0];  ; buffer_full
    CMP      r2, #0
    ADDEQ    r0, r0, #1
    BEQ      |L1.8|
    BX       lr
    ENDP
|L1.28|
    DCD      ||.data||
    AREA ||.data||, DATA, ALIGN=2
buffer_full
    DCD      0x00000000
```

In the disassembly of the nonvolatile version of the buffer loop in the above table, the statement `LDR r0, [r0, #0]` loads the value of `buffer_full` into register `r0` outside the loop labeled `|L1.12|`. Because `buffer_full` is not declared as **`volatile`**, the compiler assumes that its value cannot be modified outside the program. Having already read the value of `buffer_full` into `r0`, the compiler omits reloading the variable when optimizations are enabled, because its value cannot change. The result is the infinite loop labeled `|L1.12|`.

In contrast, in the disassembly of the volatile version of the buffer loop, the compiler assumes the value of `buffer_full` can change outside the program and performs no optimizations. Consequently, the value of `buffer_full` is loaded into register `r0` inside the loop labeled `|L1.8|`. As a result, the loop `|L1.8|` is implemented correctly in assembly code.

To avoid optimization problems caused by changes to program state external to the implementation, you must declare variables as **`volatile`** whenever their values can change unexpectedly in ways unknown to the implementation.

In practice, you must declare a variable as **`volatile`** whenever you are:

- Accessing memory-mapped peripherals.
- Sharing global variables between multiple threads.
- Accessing global variables in an interrupt routine or signal handler.

The compiler does not optimize the variables you have declared as volatile.

4.9 Code metrics

Code metrics provide a means of objectively evaluating code quality. The compiler and linker provide several facilities for generating simple code metrics and improving code quality.

In particular, you can:

- Measure code and data sizes.
- Generate dynamic callgraphs.
- Measure stack use.

Related concepts

4.10 Code metrics for measurement of code size and data size on page 4-120.

4.11 Stack use in C and C++ on page 4-121.

Related information

--info=topic[,topic,...] fromelf option.

--info=topic[,topic,...] linker option.

--map, --no_map linker option.

--callgraph, --no_callgraph linker option.

4.10 Code metrics for measurement of code size and data size

The compiler, linker, and fromelf image converter let you measure code and data size.

Use the following command-line options:

- `--info=sizes` (`armlink` and `fromelf`).
- `--info=totals` (`armcc`, `armlink`, and `fromelf`).
- `--map` (`armlink`).

Related references

7.85 --info=totals on page 7-362.

4.11 Stack use in C and C++

C and C++ both use the stack intensively.

For example, the stack holds:

- The return address of functions.
- Registers that must be preserved, as determined by the *ARM Architecture Procedure Call Standard* (AAPCS), for instance, when register contents are saved on entry into subroutines.
- Local variables, including local arrays, structures, unions, and in C++, classes.

Some stack usage is not obvious, such as:

- Local integer or floating point variables are allocated stack memory if they are spilled (that is, not allocated to a register).
- Structures are normally allocated to the stack. A space equivalent to `sizeof(struct)` padded to a multiple of four bytes is reserved on the stack. The compiler tries to allocate structures to registers instead.
- If the size of an array size is known at compile time, the compiler allocates memory on the stack. Again, a space equivalent to `sizeof(struct)` padded to a multiple of four bytes is reserved on the stack.

  Note

  Memory for variable length arrays is allocated at runtime, on the heap.

- Several optimizations can introduce new temporary variables to hold intermediate results. The optimizations include: CSE elimination, live range splitting and structure splitting. The compiler tries to allocate these temporary variables to registers. If not, it spills them to the stack.
- Generally, code compiled for processors that support only 16-bit encoded Thumb instructions makes more use of the stack than ARM code and code compiled for processors that support 32-bit encoded Thumb instructions. This is because 16-bit encoded Thumb instructions have only eight registers available for allocation, compared to fourteen for ARM code and 32-bit encoded Thumb instructions.
- The AAPCS requires that some function arguments are passed through the stack instead of the registers, depending on their type, size, and order.

Methods of estimating stack usage

Stack use is difficult to estimate because it is code dependent, and can vary between runs depending on the code path that the program takes on execution. However, it is possible to manually estimate the extent of stack utilization using the following methods:

- Link with `--callgraph` to produce a static callgraph. This shows information on all functions, including stack use.
- Link with `--info=stack` or `--info=summarystack` to list the stack usage of all global symbols.
- Use the debugger to set a watchpoint on the last available location in the stack and see if the watchpoint is ever hit.

  Note

  Running your program under a debug monitor like a *Real-Time System Model* (RTSM), in DS-5 Debugger or RealView Debugger, has a severe performance penalty, because the watched address is checked for every instruction. Using DSTREAM or RealView ICE and RealView Trace has no such penalty.

- Use the debugger, and:
  1. Allocate space in memory for the stack that is much larger than you expect to require.
  2. Fill the stack space with copies of a known value, for example, `0xDEADDEAD`.
  3. Run your application, or a fixed portion of it. Aim to use as much of the stack space as possible in the test run. For example, try to execute the most deeply nested function calls and the worst case path found by the static analysis. Try to generate interrupts where appropriate, so that they are included in the stack trace.

4. After your application has finished executing, examine the stack space of memory to see how many of the known values have been overwritten. The space has garbage in the used part and the known values in the remainder.
5. Count the number of garbage values and multiply by four, to give their size, in bytes.

The result of the calculation shows how the size of the stack has grown, in bytes.

- Use RTSM, and define a region of memory where access is not allowed directly below your stack in memory, with a map file. If the stack overflows into the forbidden region, a data abort occurs, which can be trapped by the debugger.

Methods of reducing stack usage

In general, you can lower the stack requirements of your program by:

- Writing small functions that only require a small number of variables.
- Avoiding the use of large local structures or arrays.
- Avoiding recursion, for example, by using an alternative algorithm.
- Minimizing the number of variables that are in use at any given time at each point in a function.
- Using C block scope and declaring variables only where they are required, so overlapping the memory used by distinct scopes.

The use of C block scope involves declaring variables only where they are required. This minimizes use of the stack by overlapping memory required by distinct scopes.

Note

Code performance is optimized by locating the stack in fast (zero wait-state), on-chip, 32-bit RAM. The ARM (`LDMFD` and `STMFD`) and Thumb (`PUSH` and `POP`) stack access instructions both push and pop a number of 32-bit registers on or off the stack. If the stack is in 32-bit memory, each register access takes one cycle. However, if the stack is in 16-bit memory then each register access takes two cycles, reducing overall performance.

Related information

Getting Started with DS-5, ARM DS-5 Product Overview, About Fixed Virtual Platform (FVP).

ARM DS-5 Using the Debugger.

ARM DS-5 EB FVP Reference Guide.

Fixed Virtual Platforms VE and MPS FVP Reference Guide.

Procedure Call Standard for the ARM Architecture.

--info=topic[,topic,...] fromelf option.

--info=topic[,topic,...] linker option.

--callgraph, --no_callgraph linker option.

4.12 Benefits of reducing debug information in objects and libraries

Reducing the amount of debug information in objects and libraries has a number of code size and performance benefits.

Reducing the level of debug information:

- Reduces the size of objects and libraries, thereby reducing the amount of disk space required to store them.
- Speeds up link time. In the compilation cycle, most of the link time is consumed by reading in all the debug sections and eliminating the duplicates.
- Minimizes the size of the final image. This facilitates the fast loading and processing of debug symbols by a debugger.

Related concepts

4.3 Compiler optimization levels and the debug view on page 4-108.

Related references

4.13 Methods of reducing debug information in objects and libraries on page 4-124.

4.13 Methods of reducing debug information in objects and libraries

There are a number of ways to reduce the amount of debug information being generated per source file.

For example, you can:

- Avoid conditional use of `#define` in header files. This might make it more difficult for the linker to eliminate duplicate information.
- Modify your C or C++ source files so that header files are `#included` in the same order.
- Partition header information into smaller blocks. That is, use a larger number of smaller header files rather than a smaller number of larger header files. This helps the linker to eliminate more of the common blocks.
- Only include a header file in a C or C++ source file if it is really required.
- Guard against the multiple inclusion of header files. Place multiple-inclusion guards inside the header file, rather than around the `#include` statement. For example, if you have a header file `foo.h`, add:

  ```
  #ifndef foo_h
  #define foo_h
  ...
  // rest of header file as before
  ...
  #endif /* foo_h */
  ```

  You can use the compiler option `--remarks` to warn about unguarded header files.
- Compile your code with the `--no_debug_macros` command-line option to discard preprocessor macro definitions from debug tables.
- Consider using (or not using) `--remove_unneeded_entities`.

  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.

Related concepts

2.17.1 Compilation build time on page 2-58.

4.12 Benefits of reducing debug information in objects and libraries on page 4-123.

4.3 Compiler optimization levels and the debug view on page 4-108.

Related tasks

2.17.2 Minimizing compilation build time on page 2-59.

Related references

7.34 --debug_macros, --no_debug_macros on page 7-308.

7.143 --remarks on page 7-425.

7.144 --remove_unneeded_entities, --no_remove_unneeded_entities on page 7-426.

4.14 Guarding against multiple inclusion of header files

Guarding against multiple inclusion of header files has a number of benefits.

Specifically, guarding against multiple inclusion of header files:

- Improves compilation time.
- Reduces the size of object files generated using the `-g` compiler command-line option, which can speed up link time.
- Avoids compilation errors that arise from including the same code multiple times.

For example:

```
/* foo.h */
#ifndef FOO_H
#define FOO_H 1
...
#endif
/* bar.c */
#ifndef FOO_H
#include "foo.h"
#endif
```

Related references

7.71 -g on page 7-348.

4.15 Methods of minimizing function parameter passing overhead

There are a number of ways in which you can minimize the overhead of passing parameters to functions.

For example:

- Ensure that functions take four or fewer arguments if each argument is a word or less in size. In C++, ensure that nonstatic member functions take three or fewer arguments because of the implicit `this` pointer argument that is usually passed in `R0`.
- Ensure that a function does a significant amount of work if it requires more than four arguments, so that the cost of passing the stacked arguments is outweighed.
- Put related arguments in a structure, and pass a pointer to the structure in any function call. This reduces the number of parameters and increases readability.
- Minimize the number of **`long long`** parameters, because these take two argument words that have to be aligned on an even register index.
- Minimize the number of **`double`** parameters when using software floating-point.
- Avoid functions with a variable number of parameters. Functions taking a variable number of arguments effectively pass all their arguments on the stack.

Related concepts

4.16 Returning structures from functions through registers on page 4-127.

4.16 Returning structures from functions through registers

The compiler allows functions to return structures containing multiple values through the registers, rather than the stack.

In C and C++, one way of returning multiple values from a function is to use a structure. Normally, structures are returned on the stack, with all the associated expense this entails.

To reduce memory traffic and reduce code size, the compiler enables functions to return multiple values through the registers. A function can return up to four words in a **`struct`** by qualifying the function with `__value_in_regs`. For example:

```
typedef struct s_coord { int x; int y; } coord;
coord reflect(int x1, int y1) __value_in_regs;
```

You can use `__value_in_regs` anywhere where multiple values have to be returned from a function. Examples include:

- Returning multiple values from C and C++ functions.
- Returning multiple values from embedded assembly language functions.
- Making supervisor calls.
- Re-implementing `__user_initial_stackheap`.

Related concepts

4.15 Methods of minimizing function parameter passing overhead on page 4-126.

Related references

9.19 __value_in_regs on page 9-535.

4.17 Functions that return the same result when called with the same arguments

A function that always returns the same result when called with the same arguments, and does not change any global data, is referred to as a pure function.

By definition, it is sufficient to evaluate any particular call to a pure function only once. Because the result of a call to the function is guaranteed to be the same for any identical call, each subsequent call to the function in code can be replaced with the result of the original call.

Using the keyword `__pure` when declaring a function indicates that the function is a pure function.

By definition, pure functions cannot have side effects. For example, a pure function cannot read or write global state by using global variables or indirecting through pointers, because accessing global state can violate the rule that the function must return the same value each time when called twice with the same parameters. Therefore, you must use `__pure` carefully in your programs. Where functions can be declared `__pure`, however, the compiler can often perform powerful optimizations, such as *Common Subexpression Eliminations* (CSEs).

Related references

9.31 __attribute__((const)) function attribute on page 9-550.

9.46 __attribute__((pure)) function attribute on page 9-565.

4.18 Comparison of pure and impure functions on page 4-129.

9.13 __pure on page 9-529.

4.18 Comparison of pure and impure functions

The two sample routines in the following table illustrate the use of the `__pure` keyword.

Both routines call a function `fact()` to calculate the sum of `n!` and `n!`. `fact()` depends only on its input argument n to compute `n!`. Therefore, `fact()` is a pure function.

The first routine shows a naive implementation of the function `fact`(), where `fact()` is not declared `__pure`. In the second implementation, `fact()` is qualified as `__pure` to indicate to the compiler that it is a pure function.

Table 4-7 C code for pure and impure functions

A pure function not declared __pure	A pure function declared __pure

`int fact(int n)`
`{`
`    int f = 1;`
`    while (n > 0)`
`        f *= n--;`
`    return f;`
`}`
`int foo(int n)`
`{`
`    return fact(n)+fact(n);`
`}`
	`int fact(int n) __pure`
	`{`
	`    int f = 1;`
	`    while (n > 0)`
	`        f *= n--;`
	`    return f;`
	`}`
	`int foo(int n)`
	`{`
	`    return fact(n)+fact(n);`
	`}`

The following table shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations above, where the C code for each implementation has been compiled using the option `-O2`, and inlining has been suppressed.

Table 4-8 Disassembly for pure and impure functions

A pure function not declared __pure	A pure function declared __pure

`fact PROC`
`    ...`
`foo PROC`
`    MOV      r3, r0`
`    PUSH     {lr}`
`    BL       fact`
`    MOV      r2, r0`
`    MOV      r0, r3`
`    BL       fact`
`    ADD      r0, r0, r2`
`    POP      {pc}`
`    ENDP`
	`fact PROC`
	`    ...`
	`foo PROC`
	`    PUSH     {lr}`
	`    BL       fact`
	`    LSL      r0,r0,#1`
	`    POP      {pc}`
	`    ENDP`

In the disassembly where `fact()` is not qualified as `__pure`, `fact()` is called twice because the compiler does not know that the function is a candidate for *Common Subexpression Elimination* (CSE). In contrast, in the disassembly where `fact()` is qualified as `__pure`, `fact()` is called only once, instead of twice, because the compiler has been able to perform CSE when adding `fact(n) + fact(n)`.

Related concepts

4.17 Functions that return the same result when called with the same arguments on page 4-128.

Related references

9.13 __pure on page 9-529.

4.19 Recommendation of postfix syntax when qualifying functions with ARM function modifiers

You can use function modifiers such as `__pure` either prefix or postfix, that is before the function declaration or after the parameter list. ARM recommends using the more precise postfix syntax.

Many ARM keyword extensions modify the behavior or calling sequence of a function. For example, `__pure`, `__irq`, `__swi`, `__swi_indirect` , `__softfp`, and `__value_in_regs` all behave in this way.

These function modifiers all have a common syntax. A function modifier such as `__pure` can qualify a function declaration either:

- Before the function declaration. For example:

  ```
  __pure int foo(int);
  ```
- After the closing parenthesis on the parameter list. For example:

  ```
  int foo(int) __pure;
  ```

For simple function declarations, each syntax is unambiguous. However, for a function whose return type or arguments are function pointers, the prefix syntax is imprecise. For example, the following function returns a function pointer, but it is not clear whether `__pure` modifies the function itself or its returned pointer type:

```
__pure int (*foo(int)) (int); /* declares 'foo' as a (pure?) function
                                 that returns a pointer to a (pure?)
                                 function.
                                 It is ambiguous which of the two
                                 function types is pure. */
```

In fact, the single `__pure` keyword at the front of the declaration of `foo` modifies both `foo` itself *and* the function pointer type returned by `foo`.

In contrast, the postfix syntax enables clear distinction between whether `__pure` applies to the argument, the return type, or the base function, when declaring a function whose argument and return types are function pointers. For example:

```
int (*foo1(int) __pure) (int);        /* foo1 is a pure function
                                         returning a pointer to
                                         a normal function */
int (*foo2(int)) (int) __pure;        /* foo2 is a function
                                         returning a pointer to
                                         a pure function */
int (*foo3(int) __pure) (int) __pure; /* foo3 is a pure function
                                         returning a pointer to
                                         a pure function */
```

In this example:

- `foo1` and `foo3` are modified themselves.
- `foo2` and `foo3` return a pointer to a modified function.
- The functions `foo3` and `foo` are identical.

Because the postfix syntax is more precise than the prefix syntax, ARM recommends that, where possible, you make use of the postfix syntax when qualifying functions with ARM function modifiers.

Related references

9.11 __irq on page 9-526.
9.13 __pure on page 9-529.
9.15 __softfp on page 9-531.
9.16 __svc on page 9-532.
9.17 __svc_indirect on page 9-533.
9.19 __value_in_regs on page 9-535.

4.20 Inline functions

Inline functions offer a trade-off between code size and performance. By default, the compiler decides for itself whether to inline code or not.

As a general rule, when compiling with `-Ospace`, the compiler makes sensible decisions about inlining with a view to producing code of minimal size. This is because code size for embedded systems is of fundamental importance. When compiling with `-Otime`, the compiler inlines in most cases, but still avoids large code growth. On NEON, calls to non-inline functions from within a loop inhibit vectorization, and require explicit indication that they are to be inlined for vectorization to take place.

In most circumstances, the decision to inline a particular function is best left to the compiler. However, you can give the compiler a hint that a function is required to be inlined by using the appropriate inline keyword.

Functions that are qualified with the **`__inline`**, **`inline`**, or **`__forceinline`** keywords are called inline functions. In C++, member functions that are defined inside a class, struct, or union, are also inline functions.

The compiler also offers a range of other facilities for modifying its behavior with respect to inlining. There are several factors you must take into account when deciding whether to use these facilities, or more generally, whether to inline a function at all.

The linker is able to apply some degree of function inlining to functions that are very short.

Related concepts

4.21 Compiler decisions on function inlining on page 4-132.

4.22 Automatic function inlining and static functions on page 4-133.

4.23 Inline functions and removal of unused out-of-line functions at link time on page 4-134.

4.24 Automatic function inlining and multifile compilation on page 4-135.

4.26 Compiler modes and inline functions on page 4-137.

4.27 Inline functions in C++ and C90 mode on page 4-138.

4.28 Inline functions in C99 mode on page 4-139.

4.29 Inline functions and debugging on page 4-141.

Related references

4.25 Restriction on overriding compiler decisions about function inlining on page 4-136.

7.11 --autoinline, --no_autoinline on page 7-281.

7.65 --forceinline on page 7-339.

9.6 __forceinline on page 9-520.

9.8 __inline on page 9-523.

7.86 --inline, --no_inline on page 7-363.

Related information

--inline, --no_inline linker option.

4.21 Compiler decisions on function inlining

When function inlining is enabled, the compiler uses a complex decision tree to decide if a function is to be inlined.

The following simplified algorithm is used:

1. If the function is qualified with `__forceinline`, the function is inlined if it is possible to do so.
2. If the function is qualified with `__inline` and the option `--forceinline` is selected, the function is inlined if it is possible to do so.

   If the function is qualified with `__inline` and the option `--forceinline` is not selected, the function is inlined if it is practical to do so.
3. If the optimization level is `-O2` or higher, or `--autoinline` is specified, the compiler automatically inlines functions if it is practical to do so, even if you do not explicitly give a hint that function inlining is wanted.

When deciding if it is practical to inline a function, the compiler takes into account several other criteria, such as:

- The size of the function, and how many times it is called.
- The current optimization level.
- Whether it is optimizing for speed (`-Otime`) or size (`-Ospace`).
- Whether the function has external or static linkage.
- How many parameters the function has.
- Whether the return value of the function is used.

Ultimately, the compiler can decide not to inline a function, even if the function is qualified with `__forceinline`. As a general rule:
- Smaller functions stand a better chance of being inlined.
- Compiling with `-Otime` increases the likelihood that a function is inlined.
- Large functions are not normally inlined because this can adversely affect code density and performance.

A recursive function is never inlined into itself, even if `__forceinline` is used.

Related concepts

4.20 Inline functions on page 4-131.

4.22 Automatic function inlining and static functions on page 4-133.

4.23 Inline functions and removal of unused out-of-line functions at link time on page 4-134.

4.24 Automatic function inlining and multifile compilation on page 4-135.

4.26 Compiler modes and inline functions on page 4-137.

4.27 Inline functions in C++ and C90 mode on page 4-138.

4.28 Inline functions in C99 mode on page 4-139.

4.29 Inline functions and debugging on page 4-141.

Related references

4.25 Restriction on overriding compiler decisions about function inlining on page 4-136.

7.11 --autoinline, --no_autoinline on page 7-281.

7.65 --forceinline on page 7-339.

9.6 __forceinline on page 9-520.

9.8 __inline on page 9-523.

7.86 --inline, --no_inline on page 7-363.

7.119 -Onum on page 7-399.

7.124 -Ospace on page 7-406.

7.125 -Otime on page 7-407.

4.22 Automatic function inlining and static functions

At `-O2` and `-O3` levels of optimization, or when `--autoinline` is specified, the compiler can automatically inline functions if it is practical and possible to do so, even if the functions are not declared as `__inline` or **`inline`**.

This works best for static functions, because if all use of a static function can be inlined, no out-of-line copy is required. Unless a function is explicitly declared as **`static`** (or `__inline`), the compiler has to retain the out-of-line version of it in the object file in case it is called from some other module.

It is best to mark all non-inline functions as static if they are not used outside the translation unit where they are defined (a translation unit being the preprocessed output of a source file together with all of the headers and source files included as a result of the `#include` directive). Typically, you do not want to place definitions of non-inline functions in header files.

If you fail to declare functions that are never called from outside a module as **`static`**, code can be adversely affected. In particular, you might have:

- A larger code size, because out-of-line versions of functions are retained in the image.

  When a function is automatically inlined, both the in-line version *and* an out-of-line version of the function might end up in the final image, unless the function is declared as **`static`**. This might increase code size.
- An unnecessarily complicated debug view, because there are both inline versions and out-of-line versions of functions to display.

  Retaining both inline and out-of-line copies of a function in code can sometimes be confusing when setting breakpoints or single-stepping in a debug view. The debugger has to display both in-line and out-of-line versions in its interleaved source view so that you can see what is happening when stepping through either the in-line or out-of-line version.

Because of these problems, declare non-inline functions as **`static`** when you are sure that they can never be called from another module.

Related concepts

4.20 Inline functions on page 4-131.
4.21 Compiler decisions on function inlining on page 4-132.
4.23 Inline functions and removal of unused out-of-line functions at link time on page 4-134.
4.24 Automatic function inlining and multifile compilation on page 4-135.
4.26 Compiler modes and inline functions on page 4-137.
4.27 Inline functions in C++ and C90 mode on page 4-138.
4.28 Inline functions in C99 mode on page 4-139.
4.29 Inline functions and debugging on page 4-141.

Related references

4.25 Restriction on overriding compiler decisions about function inlining on page 4-136.
7.11 --autoinline, --no_autoinline on page 7-281.
7.119 -Onum on page 7-399.

4.23 Inline functions and removal of unused out-of-line functions at link time

The linker cannot remove unused out-of-line functions from an object unless you place the unused out-of-line functions in their own sections.

Use one of the following methods to place unused out-of-line functions in their own sections:

- `--split_sections`.
- `__attribute__((section("name")))`.
- `#pragma arm section [section_type_list]`.
- Linker feedback.

`--feedback` is typically an easier method of enabling unused function removal.

Related concepts

4.20 Inline functions on page 4-131.

4.21 Compiler decisions on function inlining on page 4-132.

4.22 Automatic function inlining and static functions on page 4-133.

4.24 Automatic function inlining and multifile compilation on page 4-135.

4.26 Compiler modes and inline functions on page 4-137.

4.27 Inline functions in C++ and C90 mode on page 4-138.

4.28 Inline functions in C99 mode on page 4-139.

4.29 Inline functions and debugging on page 4-141.

Related references

4.25 Restriction on overriding compiler decisions about function inlining on page 4-136.

7.62 --feedback=filename on page 7-336.

7.155 --split_sections on page 7-438.

9.67 __attribute__((section("name"))) variable attribute on page 9-586.

9.77 #pragma arm section [section_type_list] on page 9-596.

4.24 Automatic function inlining and multifile compilation

If you are compiling with the `--multifile` option, the compiler can perform automatic inlining for calls to functions that are defined in other translation units.

In RVCT 4.0 the `--multifile` option is enabled by default at `-O3` level.

In ARM Compiler 4.1 and later the `--multifile` option is disabled by default, regardless of the optimization level.

For `--multifile`, both translation units must be compiled in the same invocation of the compiler.

Related concepts

4.20 Inline functions on page 4-131.
4.21 Compiler decisions on function inlining on page 4-132.
4.22 Automatic function inlining and static functions on page 4-133.
4.23 Inline functions and removal of unused out-of-line functions at link time on page 4-134.
4.26 Compiler modes and inline functions on page 4-137.
4.27 Inline functions in C++ and C90 mode on page 4-138.
4.28 Inline functions in C99 mode on page 4-139.
4.29 Inline functions and debugging on page 4-141.

Related references

4.25 Restriction on overriding compiler decisions about function inlining on page 4-136.
7.11 --autoinline, --no_autoinline on page 7-281.
7.86 --inline, --no_inline on page 7-363.
7.114 --multifile, --no_multifile on page 7-393.
7.119 -Onum on page 7-399.

4.25 Restriction on overriding compiler decisions about function inlining

You can enable and disable function inlining, but you cannot override decisions the compiler makes about when it is practical to inline a function.

For example, you cannot force a function to be inlined if the compiler thinks it is not sensible to do so. Even if you use `--forceinline` or `__forceinline`, the compiler only inlines functions if it is possible to do so.

Related concepts

4.20 Inline functions on page 4-131.

4.21 Compiler decisions on function inlining on page 4-132.

4.22 Automatic function inlining and static functions on page 4-133.

4.23 Inline functions and removal of unused out-of-line functions at link time on page 4-134.

4.24 Automatic function inlining and multifile compilation on page 4-135.

4.26 Compiler modes and inline functions on page 4-137.

4.27 Inline functions in C++ and C90 mode on page 4-138.

4.28 Inline functions in C99 mode on page 4-139.

4.29 Inline functions and debugging on page 4-141.

Related references

7.11 --autoinline, --no_autoinline on page 7-281.

7.65 --forceinline on page 7-339.

9.6 __forceinline on page 9-520.

9.8 __inline on page 9-523.

7.86 --inline, --no_inline on page 7-363.

4.26 Compiler modes and inline functions

Compiler modes affect the behavior of inline functions.

ARM provides information about inline functions in C++, C90, and C99 modes.

The GNU Compiler Collection (GCC) web site provides information about inline functions in GNU C90 mode.

Related concepts

4.20 Inline functions on page 4-131.
4.21 Compiler decisions on function inlining on page 4-132.
4.22 Automatic function inlining and static functions on page 4-133.
4.23 Inline functions and removal of unused out-of-line functions at link time on page 4-134.
4.24 Automatic function inlining and multifile compilation on page 4-135.
4.27 Inline functions in C++ and C90 mode on page 4-138.
4.28 Inline functions in C99 mode on page 4-139.
4.29 Inline functions and debugging on page 4-141.

Related references

4.25 Restriction on overriding compiler decisions about function inlining on page 4-136.

4.27 Inline functions in C++ and C90 mode

The **`inline`** keyword is not available in C90. The effect of **`__inline`** in C90, and **`__inline`** and **`inline`** in C++, is identical.

When declaring an `extern` function to be inline, you must define it in every translation unit that it is used in. You must ensure that you use the same definition in each translation unit.

The requirement of defining the function in every translation unit applies even though it has external linkage.

If an inline function is used by more than one translation unit, its definition is typically placed in a header file.

ARM does not recommend placing definitions of non-inline functions in header files, because this can result in the creation of a separate function in each translation unit. If the non-inline function is an **`extern`** function, this leads to duplicate symbols at link time. If the non-inline function is **`static`**, this can lead to unwanted code duplication.

Member functions defined within a C++ structure, class, or union declaration, are implicitly inline. They are treated as if they are declared with the **`inline`** or **`__inline`** keyword.

Inline functions have **`extern`** linkage unless they are explicitly declared **`static`**. If an inline function is declared to be static, any out-of-line copies of the function must be unique to their translation unit, so declaring an inline function to be static could lead to unwanted code duplication.

The compiler generates a regular call to an out-of-line copy of a function when it cannot inline the function, and when it decides not to inline it.

The requirement of defining a function in every translation unit it is used in means that the compiler is not required to emit out-of-line copies of all **`extern`** inline functions. When the compiler does emit out-of-line copies of an **`extern`** inline function, it uses Common Groups, so that the linker eliminates duplicates, keeping at most one copy in the same out-of-line function from different object files.

Related concepts

4.20 Inline functions on page 4-131.
4.21 Compiler decisions on function inlining on page 4-132.
4.22 Automatic function inlining and static functions on page 4-133.
4.23 Inline functions and removal of unused out-of-line functions at link time on page 4-134.
4.24 Automatic function inlining and multifile compilation on page 4-135.
4.26 Compiler modes and inline functions on page 4-137.
4.28 Inline functions in C99 mode on page 4-139.
4.29 Inline functions and debugging on page 4-141.

Related references

4.25 Restriction on overriding compiler decisions about function inlining on page 4-136.
7.86 --inline, --no_inline on page 7-363.
9.8 __inline on page 9-523.

Related information

Elimination of common groups or sections.

4.28 Inline functions in C99 mode

The rules for C99 inline functions with external linkage differ from those of C++.

C99 distinguishes between inline definitions and external definitions. Within a given translation unit where the inline function is defined, if the inline function is always declared with **inline** and never with **extern**, it is an inline definition. Otherwise, it is an external definition. These inline definitions do not generate out-of-line copies, even when `--no_inline` is used.

Each use of an inline function might be inlined using a definition from the same translation unit (that might be an inline definition or an external definition), or it might become a call to an external definition. If an inline function is used, it must have exactly one external definition in some translation unit. This is the same rule that applies to using any external function. In practise, if all uses of an inline function are inlined, no error occurs if the external definition is missing. If you use `--no_inline`, only external definitions are used.

Typically, you put inline functions with external linkage into header files as inline definitions, using **inline**, and not using **extern**. There is also an external definition in one source file. For example:

```
/* example_header.h */
inline int my_function (int i)
{
   return i + 42; // inline definition
}
/* file1.c */
#include "example_header.h"
   ... // uses of my_function()
/* file2.c */
#include "example_header.h"
   ... // uses of my_function()
/* myfile.c */
#include "example_header.h"
extern inline int my_function(int); // causes external definition.
```

This is the same strategy that is typically used for C++, but in C++ there is no special external definition, and no requirement for it.

The definitions of inline functions can be different in different translation units. However, in typical use, as in the above example, they are identical.

When compiling with `--multifile`, calls in one translation unit might be inlined using the external definition in another translation unit.

C99 places some restrictions on inline definitions. They cannot define modifiable local static objects. They cannot reference identifiers with static linkage.

In C99 mode, as with all other modes, the effects of **__inline** and **inline** are identical.

Inline functions with static linkage have the same behavior in C99 as in C++.

Related concepts

4.20 Inline functions on page 4-131.
4.21 Compiler decisions on function inlining on page 4-132.
4.22 Automatic function inlining and static functions on page 4-133.
4.23 Inline functions and removal of unused out-of-line functions at link time on page 4-134.
4.24 Automatic function inlining and multifile compilation on page 4-135.
4.26 Compiler modes and inline functions on page 4-137.
4.27 Inline functions in C++ and C90 mode on page 4-138.
4.29 Inline functions and debugging on page 4-141.

Related references

4.25 Restriction on overriding compiler decisions about function inlining on page 4-136.
7.86 --inline, --no_inline on page 7-363.

7.114 --multifile, --no_multifile on page 7-393.

9.8 __inline on page 9-523.

4.29 Inline functions and debugging

The debug view generated for inline functions is generally good. However, it is sometimes useful to avoid inlining functions because in some situations, debugging is clearer if they are not inlined.

You can enable and disable the inlining of functions using the `--no_inline`, `--inline`, `--autoinline` and `--no_autoinline` command-line options.

The debug view can also be adversely affected by retaining both inline and out-of-line copies of a function when out-of-line copies are not required. Functions that are never called from outside a module can be declared as static functions to avoid an unnecessarily complicated debug view.

Related concepts

4.20 Inline functions on page 4-131.
4.21 Compiler decisions on function inlining on page 4-132.
4.22 Automatic function inlining and static functions on page 4-133.
4.23 Inline functions and removal of unused out-of-line functions at link time on page 4-134.
4.24 Automatic function inlining and multifile compilation on page 4-135.
4.26 Compiler modes and inline functions on page 4-137.
4.27 Inline functions in C++ and C90 mode on page 4-138.
4.28 Inline functions in C99 mode on page 4-139.

Related references

4.25 Restriction on overriding compiler decisions about function inlining on page 4-136.
7.11 --autoinline, --no_autoinline on page 7-281.
7.65 --forceinline on page 7-339.
9.6 __forceinline on page 9-520.
9.8 __inline on page 9-523.
7.86 --inline, --no_inline on page 7-363.

Related information

--inline, --no_inline linker option.

4.30 Types of data alignment

All access to data in memory can be classified into a number of different categories.

These categories are as follows:

- Natural alignment, for example, on a word boundary at 0x1004. The ARM compiler normally aligns variables and pads structures so that these items are accessed efficiently using `LDR` and `STR` instructions.
- Known but non-natural alignment, for example, a word at address 0x1001. This type of alignment commonly occurs when structures are packed to remove unnecessary padding. In C and C++, the `__packed` qualifier or the `#pragma pack(n)` pragma let you signify that a structure is packed.
- Unknown alignment, for example, a word at an arbitrary address. This type of alignment commonly occurs when defining a pointer that can point to a word at any address. In C and C++, the `__packed` qualifier or the `#pragma pack(n)` pragma let you signify that a pointer can access a word on a non-natural alignment boundary.

Related concepts

4.31 Advantages of natural data alignment on page 4-143.

4.34 Unaligned data access in C and C++ code on page 4-146.

4.35 The __packed qualifier and unaligned data access in C and C++ code on page 4-147.

Related references

4.32 Compiler storage of data objects by natural byte alignment on page 4-144.

4.33 Relevance of natural data alignment at compile time on page 4-145.

9.95 #pragma pack(n) on page 9-615.

4.31 Advantages of natural data alignment

The various C data types are aligned on specific byte boundaries to maximize storage potential and to provide for fast, efficient memory access with the ARM instruction set.

For example, the ARM architecture can access a four-byte variable using only one instruction when the object is stored at an address divisible by four, so four-byte objects are located on four-byte boundaries.

ARM and Thumb processors are designed to efficiently access *naturally aligned* data, that is, doublewords that lie on addresses that are multiples of eight, words that lie on addresses that are multiples of four, halfwords that lie on addresses that are multiples of two, and single bytes that lie at any byte address. Such data is located on its natural size boundary.

Related concepts

4.30 Types of data alignment on page 4-142.

4.34 Unaligned data access in C and C++ code on page 4-146.

4.35 The __packed qualifier and unaligned data access in C and C++ code on page 4-147.

Related references

4.32 Compiler storage of data objects by natural byte alignment on page 4-144.

4.33 Relevance of natural data alignment at compile time on page 4-145.

4.32 Compiler storage of data objects by natural byte alignment

C data types are aligned on specific byte boundaries, depending on their type.

By default, the compiler stores data objects by byte alignment as shown in the following table.

Table 4-9 Compiler storage of data objects by byte alignment

Type	Bytes	Alignment
`char`, `bool`, `_Bool`	1	Located at any byte address.
`short`, `wchar_t`	2	Located at any address that is evenly divisible by 2.
`float`, `int`, `long`, `pointer`	4	Located at an address that is evenly divisible by 4.
`long long`, `double`, `long double`	8	Located at an address that is evenly divisible by 8.

Related concepts

4.30 Types of data alignment on page 4-142.

4.31 Advantages of natural data alignment on page 4-143.

4.34 Unaligned data access in C and C++ code on page 4-146.

4.35 The __packed qualifier and unaligned data access in C and C++ code on page 4-147.

Related references

4.33 Relevance of natural data alignment at compile time on page 4-145.

4.33 Relevance of natural data alignment at compile time

Data alignment becomes relevant when the compiler allocates memory locations to variables.

For example, in the following structure, a three-byte gap is required between `bmem` and `cmem`.

```
struct example_st {
  int amem;
  char bmem;
  int cmem;
};
```

Related concepts

4.30 Types of data alignment on page 4-142.

4.31 Advantages of natural data alignment on page 4-143.

4.34 Unaligned data access in C and C++ code on page 4-146.

4.35 The __packed qualifier and unaligned data access in C and C++ code on page 4-147.

Related references

4.32 Compiler storage of data objects by natural byte alignment on page 4-144.

4.34 Unaligned data access in C and C++ code

It can be necessary to access unaligned data in memory, for example, when porting legacy code from a CISC architecture where instructions are available to directly access unaligned data in memory.

On ARMv4 and ARMv5 architectures, and on the ARMv6 architecture depending on how it is configured, care is required when accessing unaligned data in memory, to avoid unexpected results. For example, when C or C++ source code uses a conventional pointer to read a word in C or C++ source code, the ARM compiler generates assembly language code that reads the word using an `LDR` instruction. This works as expected when the address is a multiple of four, for example if it lies on a word boundary. However, if the address is not a multiple of four, the `LDR` instruction returns a rotated result rather than performing a true unaligned word load. Generally, this rotation is not what the programmer expects.

On ARMv6 and later architectures, unaligned access is fully supported.

Related concepts

4.30 Types of data alignment on page 4-142.

4.31 Advantages of natural data alignment on page 4-143.

4.35 The __packed qualifier and unaligned data access in C and C++ code on page 4-147.

Related references

4.32 Compiler storage of data objects by natural byte alignment on page 4-144.

4.33 Relevance of natural data alignment at compile time on page 4-145.

4.35 The __packed qualifier and unaligned data access in C and C++ code

The `__packed` qualifier sets the alignment of any valid type to 1.

This enables objects of packed type to be read or written using unaligned access.

Examples of objects that can be packed include:

- Structures.
- Unions.
- Pointers.

Related concepts

4.30 Types of data alignment on page 4-142.

4.31 Advantages of natural data alignment on page 4-143.

4.34 Unaligned data access in C and C++ code on page 4-146.

4.36 Unaligned fields in structures on page 4-148.

4.37 Performance penalty associated with marking whole structures as packed on page 4-149.

Related references

4.32 Compiler storage of data objects by natural byte alignment on page 4-144.

4.33 Relevance of natural data alignment at compile time on page 4-145.

9.12 __packed on page 9-527.

9.95 #pragma pack(n) on page 9-615.

4.36 Unaligned fields in structures

You can use the `__packed` qualifier to create unaligned fields in structures. This saves space because the compiler does not need to pad fields to their natural size boundary.

For efficiency, fields in a structure are positioned on their natural size boundary. This means that the compiler often inserts padding between fields to ensure that they are naturally aligned.

When space is at a premium, you can use the `__packed` qualifier to create structures without padding between fields. Structures can be packed in the following ways:

- The entire **`struct`** can be declared as `__packed`. For example:

  ```
  __packed struct mystruct
  {
      char c;
      short s;
  } // not recommended
  ```

  Each field of the structure inherits the `__packed` qualifier.

  Declaring an entire **`struct`** as `__packed` typically incurs a penalty both in code size and performance.
- Individual non-aligned fields within the **`struct`** can be declared as `__packed`. For example:

  ```
  struct mystruct
  {
      char c;
      __packed short s; // recommended
  }
  ```

  This is the recommended approach to packing structures.

Note

The same principles apply to unions. You can declare either an entire union as `__packed`, or use the `__packed` attribute to identify components of the union that are unaligned in memory.

Related concepts

4.35 The __packed qualifier and unaligned data access in C and C++ code on page 4-147.

4.37 Performance penalty associated with marking whole structures as packed on page 4-149.

4.40 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 4-152.

Related references

9.12 __packed on page 9-527.

9.95 #pragma pack(n) on page 9-615.

4.37 Performance penalty associated with marking whole structures as packed

Reading from and writing to whole structures qualified with `__packed` requires unaligned accesses and can therefore incur a performance penalty.

When optimizing a **`struct`** that is packed, the compiler tries to deduce the alignment of each field, to improve access. However, it is not always possible for the compiler to deduce the alignment of each field in a `__packed` **`struct`**. In contrast, when individual fields in a **`struct`** are declared as `__packed`, fast access is guaranteed to naturally aligned members within the **`struct`**. Therefore, when the use of a packed structure is required, ARM recommends that you always pack individual fields of the structure, rather than the entire structure itself.

Note

Declaring individual non-aligned fields of a **`struct`** as `__packed` also has the advantage of making it clearer to the programmer which fields of the **`struct`** are not naturally aligned.

Related concepts

4.35 The __packed qualifier and unaligned data access in C and C++ code on page 4-147.

4.36 Unaligned fields in structures on page 4-148.

4.40 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 4-152.

Related references

9.12 __packed on page 9-527.

9.95 #pragma pack(n) on page 9-615.

4.38 Unaligned pointers in C and C++ code

If you want to define a pointer that can point to a word at any address, you must specify the `__packed` qualifier.

By default, the compiler expects conventional C and C++ pointers to point to naturally aligned words in memory because this enables the compiler to generate more efficient code.

For example, to specify an unaligned pointer:

```
__packed int *pi; // pointer to unaligned int
```

When a pointer is declared as `__packed`, the compiler generates code that correctly accesses the dereferenced value of the pointer, regardless of its alignment. The generated code consists of a sequence of byte accesses, or variable alignment-dependent shifting and masking instructions, rather than a simple `LDR` instruction. Consequently, declaring a pointer as `__packed` incurs a performance and code size penalty.

Related concepts

4.39 Unaligned Load Register (LDR) instructions generated by the compiler on page 4-151.

Related references

9.12 __packed on page 9-527.

7.164 --unaligned_access, --no_unaligned_access on page 7-448.

4.39 Unaligned Load Register (LDR) instructions generated by the compiler

In some circumstances, where it is legal to do so, the compiler might intentionally generate unaligned `LDR` instructions.

In particular, the compiler can do this to load halfwords from memory, even where the architecture supports dedicated halfword load instructions.

For example, to access an unaligned **`short`** within a `__packed` structure, the compiler might load the required halfword into the top half of a register and then shift it down to the bottom half. This operation requires only one memory access, whereas performing the same operation using `LDRB` instructions requires two memory accesses, plus instructions to merge the two bytes.

Related concepts

4.38 Unaligned pointers in C and C++ code on page 4-150.

Related references

9.12 __packed on page 9-527.

7.164 --unaligned_access, --no_unaligned_access on page 7-448.

4.40 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

These comparisons illustrate the differences between the methods of packing structures.

Comparison of an unpacked `struct`, a __packed `struct`, and a `struct` with individually __packed fields

The differences between not packing a **`struct`**, packing an entire **`struct`**, and packing individual fields of a **`struct`** are illustrated by the three implementations of a **`struct`** shown in the following table.

Table 4-10 C code for an unpacked struct, a packed struct, and a struct with individually packed fields

Unpacked struct	__packed struct	__packed fields

`struct foo`
`{`
`    char one;`
`    short two;`
`    char three;`
`    int four;`
`} c;`
	`__packed struct foo`
	`{`
	`    char one;`
	`    short two;`
	`    char three;`
	`    int four;`
	`} c;`
		`struct foo`
		`{`
		`    char one;`
		`    __packed short two;`
		`    char three;`
		`    int four;`
		`} c;`

In the first implementation, the **`struct`** is not packed. In the second implementation, the entire structure is qualified as `__packed`. In the third implementation, the `__packed` attribute is removed from the structure and the individual field that is not naturally aligned is declared as `__packed`.

The following table shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations of the preceding table, where the C code for each implementation has been compiled using the option `-O2`.

Table 4-11 Disassembly for an unpacked struct, a packed struct, and a struct with individually packed fields

Unpacked struct	__packed struct	__packed fields

`; r0 contains address of c`
`; char one`
`LDRB    r1, [r0, #0]`
`; short two`
`LDRSH   r2, [r0, #2]`
`; char three`
`LDRB    r3, [r0, #4]`
`; int four`
`LDR     r12, [r0, #8]`
	`; r0 contains address of c`
	`; char one`
	`LDRB  r1, [r0, #0]`
	`; short two`
	`LDRB  r2, [r0, #1]`
	`LDRSB r12, [r0, #2]`
	`ORR   r2, r12, r2, LSL #8`
	`; char three`
	`LDRB  r3, [r0, #3]`
	`; int four`
	`ADD   r0, r0, #4`
	`BL    __aeabi_uread4`
		`; r0 contains address of c`
		`; char one`
		`LDRB  r1, [r0, #0]`
		`; short two`
		`LDRB  r2, [r0, #1]`
		`LDRSB r12, [r0, #2]`
		`ORR   r2, r12, r2, LSL #8`
		`; char three`
		`LDRB  r3, [r0, #3]`
		`; int four`
		`LDR   r12, [r0, #4]`

Note

The `-Ospace` and `-Otime` compiler options control whether accesses to unaligned elements are made inline or through a function call. Using `-Otime` results in inline unaligned accesses. Using `-Ospace` results in unaligned accesses made through function calls.

In the disassembly of the unpacked **`struct`** example above, the compiler always accesses data on aligned word or halfword addresses. The compiler is able to do this because the **`struct`** is padded so that every member of the **`struct`** lies on its natural size boundary.

In the disassembly of the `__packed` **`struct`** example above, fields `one` and `three` are aligned on their natural size boundaries by default, so the compiler makes aligned accesses. The compiler always carries out aligned word or halfword accesses for fields it can identify as being aligned. For the unaligned field `two`, the compiler uses multiple aligned memory accesses (`LDR`/`STR`/`LDM`/`STM`), combined with fixed shifting and masking, to access the correct bytes in memory. The compiler calls the *ARM Embedded Application Binary Interface* (AEABI) runtime routine `__aeabi_uread4` for reading an unsigned word at an unknown alignment to access field `four` because it is not able to determine that the field lies on its natural size boundary.

In the disassembly of the **`struct`** with individually packed fields example above, fields `one`, `two`, and `three` are accessed in the same way as in the case where the entire **`struct`** is qualified as `__packed`. In contrast to the situation where the entire **`struct`** is packed, however, the compiler makes a word-aligned access to the field `four`. This is because the presence of the `__packed short` within the structure helps the compiler to determine that the field `four` lies on its natural size boundary.

Comparison of a __packed `struct` and a #pragma packed `struct`

The differences between a `__packed` **`struct`** and a `#pragma packed` **`struct`** are illustrated by the two implementations of a **`struct`** shown in the following table.

Table 4-12 C code for a packed struct and a pragma packed struct

__packed struct	#pragma packed struct
`__packed struct foobar`	`#pragma push`
`{`	`#pragma pack(1)`
`    char x;`	`struct foobar`
`    short y[10];`	`{`
`};`	`    char x;`
`short get_y0(struct foobar *s)`	`    short y[10];`
`{`	`};`
`    // Unaligned-capable load`	`#pragma pop`
`    return *s->y;`	`short get_y0(struct foobar *s)`
`}`	`{`
`short *get_y(struct foobar *s)`	`    // Unaligned-capable load`
`{`	`    return *s->y;`
`    return s->y;    // Compile error`	`}`
`}`	`short *get_y(struct foobar *s)`
	`{`
	`    return s->y;    // No error`
	`    // Potentially illegal unaligned load,`
	`    // depending on use of result`
	`}`

In the first implementation, taking the address of a field in a `__packed` **`struct`** or a `__packed` field in a **`struct`** yields a `__packed` pointer, and the compiler generates a type error if you try to implicitly cast this to a non-`__packed` pointer. In the second implementation, in contrast, taking the address of a field in a `#pragma packed` **`struct`** does not yield a `__packed`-qualified pointer. However, the field might not be properly aligned for its type, and dereferencing such an unaligned pointer results in Undefined behavior.

Related concepts

4.36 Unaligned fields in structures on page 4-148.

4.37 Performance penalty associated with marking whole structures as packed on page 4-149.

Related references

7.124 -Ospace on page 7-406.

7.125 -Otime on page 7-407.

9.12 __packed on page 9-527.

9.58 __attribute__((packed)) type attribute on page 9-577.

9.66 __attribute__((packed)) variable attribute on page 9-585.

9.95 #pragma pack(n) on page 9-615.

Related information

Application Binary Interface (ABI) for the ARM Architecture.

4.41 Compiler support for floating-point arithmetic

The compiler provides many features for managing floating-point arithmetic both in hardware and in software.

For example, you can specify software or hardware support for floating-point, particular hardware architectures, and the level of conformance to IEEE floating-point standards.

The selection of floating-point options determines various trade-offs between floating-point performance, system cost, and system flexibility. To obtain the best trade-off between performance, cost, and flexibility, you have to make sensible choices in your selection of floating-point options.

Floating-point arithmetic can be supported, either:

- In software, through the floating-point library `fplib`. This library provides functions that can be called to implement floating-point operations using no additional hardware.
- In hardware, using a hardware *Vector Floating Point* (VFP) coprocessor with the ARM processor to provide the required floating-point operations. VFP is a coprocessor architecture that implements IEEE floating-point and supports single and double precision, but not extended precision.

  Note

  In practice, floating-point arithmetic in the VFP is implemented using a combination of hardware, that executes the common cases, and software, that deals with the uncommon cases, and cases causing exceptions.

Code that uses hardware support for floating-point arithmetic is more compact and offers better performance than code that performs floating-point arithmetic in software. However, hardware support for floating-point arithmetic requires a VFP coprocessor.

Related concepts

4.42 Default selection of hardware or software floating-point support on page 4-156.
4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.
4.44 Vector Floating-Point (VFP) architectures on page 4-159.
4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.
4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.
4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.
4.48 Half-precision floating-point number format on page 4-164.
4.49 Compiler support for floating-point computations and linkage on page 4-165.
4.50 Types of floating-point linkage on page 4-166.
4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.
4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.
9.98 #pragma softfp_linkage, #pragma no_softfp_linkage on page 9-619.
7.29 --cpu=name compiler option on page 7-302.
9.116 __fabs intrinsic on page 9-639.
7.66 --fp16_format=format on page 7-340.
7.67 --fpmode=model on page 7-341.
7.68 --fpu=list on page 7-343.
7.69 --fpu=name compiler option on page 7-344.
9.139 __sqrt intrinsic on page 9-665.
9.157 GNU built-in functions on page 9-689.
9.158 Predefined macros on page 9-697.

9.153 VFP status intrinsic on page 9-682.

8.14 Hexadecimal floats on page 8-479.

8.38 Hexadecimal floating-point constants on page 8-503.

17.3 Limits for floating-point numbers on page 17-862.

Related information

Institute of Electrical and Electronics Engineers.

Floating-point Support.

4.42 Default selection of hardware or software floating-point support

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

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

If a VFP coprocessor is present, VFP instructions are generated. If there is no VFP coprocessor, the compiler generates code that makes calls to the software floating-point library `fplib` to carry out floating-point operations. `fplib` is available as part of the standard distribution of the ARM compilation tools suite of C libraries.

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.

4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.

4.44 Vector Floating-Point (VFP) architectures on page 4-159.

4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.

4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.

4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.

4.48 Half-precision floating-point number format on page 4-164.

4.49 Compiler support for floating-point computations and linkage on page 4-165.

4.50 Types of floating-point linkage on page 4-166.

4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.

4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.

Related information

Floating-point Support.

4.43 Example of hardware and software support differences for floating-point arithmetic

This example shows how the compiler deals with floating-point arithmetic for different processors supporting either hardware or software floating-point arithmetic.

The following example shows a function implementing floating-point arithmetic in C code.

```
float foo(float num1, float num2)
{
    float temp, temp2;
    temp = num1 + num2;
    temp2 = num2 * num2;
    return temp2 - temp;
}
```

When the example C code is compiled with the command-line options `--cpu 5TE` and `--fpu softvfp`, the compiler produces machine code with the disassembly shown below. In this case, floating-point arithmetic is performed in software through calls to library routines such as `__aeabi_fmul`.

```
||foo|| PROC
    PUSH     {r4-r6, lr}
    MOV      r4, r1
    BL       __aeabi_fadd
    MOV      r5, r0
    MOV      r1, r4
    MOV      r0, r4
    BL       __aeabi_fmul
    MOV      r1, r5
    POP      {r4-r6, lr}
    B        __aeabi_fsub
    ENDP
```

However, when the example C code is compiled with the command-line option `--fpu vfp`, the compiler produces machine code with the disassembly shown below. In this case, floating-point arithmetic is performed in hardware through floating-point arithmetic instructions such as `VMUL.F32`.

```
||foo|| PROC
    VADD.F32 s2, s0, s1
    VMUL.F32 s0, s1, s1
    VSUB.F32 s0, s0, s2
    BX       lr
    ENDP
```

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.
4.42 Default selection of hardware or software floating-point support on page 4-156.
4.44 Vector Floating-Point (VFP) architectures on page 4-159.
4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.
4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.
4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.
4.48 Half-precision floating-point number format on page 4-164.
4.49 Compiler support for floating-point computations and linkage on page 4-165.
4.50 Types of floating-point linkage on page 4-166.
4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.
4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.
7.29 --cpu=name compiler option on page 7-302.
7.28 --cpu=list on page 7-301.
7.68 --fpu=list on page 7-343.
7.69 --fpu=name compiler option on page 7-344.

Related information

Application Binary Interface (ABI) for the ARM Architecture.

4.44 Vector Floating-Point (VFP) architectures

ARM supports several versions of the VFP architecture, implemented in different ARM architectures.

VFP architectures provide both single and double precision operations. Many operations can take place in either scalar form or in vector form. Several versions of the architecture are supported, including:

- VFPv2, implemented in:
  — VFP9-S, available as a separately licensable option for the ARM926E, ARM946E and ARM966E processors.
- VFPv3, implemented on ARM architecture v7 and later. VFPv3 is backwards compatible with VFPv2, except that it cannot trap floating point exceptions. It requires no software support code. VFPv3 has 32 double-precision registers.
- VFPv3_fp16, VFPv3 with half-precision extensions. These extensions provide conversion functions between half-precision floating-point numbers and single-precision floating-point numbers, in both directions. They can be implemented with any VFP implementation that supports single-precision floating-point numbers.
- VFPv3-D16, an implementation of VFPv3 that provides 16 double-precision registers. It is implemented on ARM architecture v7 processors that support VFP without NEON technology.
- VFPv3U, an implementation of VFPv3 that can trap floating-point exceptions. It requires software support code.
- VFPv4, implemented on ARM architecture v7 and later. VFPv4 has 32 double-precision registers. VFPv4 adds both half-precision extensions and fused multiply-add instructions to the features of VFPv3.
- VFPv4-D16, an implementation of VFPv4 that provides 16 double-precision registers. It is implemented on ARM architecture v7 processors that support VFP without NEON technology.
- VFPv4U, an implementation of VFPv4 that can trap floating-point exceptions. It requires software support code.

Note

Particular implementations of the VFP architecture might provide additional implementation-specific functionality. For example, the VFP coprocessor hardware might include extra registers for describing exceptional conditions. This extra functionality is known as *sub-architecture* functionality.

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.

4.42 Default selection of hardware or software floating-point support on page 4-156.

4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.

4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.

4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.

4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.

4.48 Half-precision floating-point number format on page 4-164.

4.49 Compiler support for floating-point computations and linkage on page 4-165.

4.50 Types of floating-point linkage on page 4-166.

4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.

4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.

Related information

ARM Application Note 133 - Using VFP with RVDS.

4.45 Limitations on hardware handling of floating-point arithmetic

ARM *Vector Floating-Point* (VFP) coprocessors are optimized to process well-defined floating-point code in hardware. Arithmetic operations that occur too rarely, or that are too complex, are not handled in hardware.

Instead, processing of these cases must be handled in software. This approach minimizes the amount of coprocessor hardware required and reduces costs.

Code provided to handle cases the VFP hardware is unable to process is known as VFP support code. When the VFP hardware is unable to deal with a situation directly, it bounces the case to VFP support code for more processing. For example, VFP support code might be called to process any of the following:

- Floating-point operations involving NaNs.
- Floating-point operations involving denormals.
- Floating-point overflow.
- Floating-point underflow.
- Inexact results.
- Division-by-zero errors.
- Invalid operations.

When support code is in place, the VFP supports a fully IEEE 754-compliant floating-point model.

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.

4.42 Default selection of hardware or software floating-point support on page 4-156.

4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.

4.44 Vector Floating-Point (VFP) architectures on page 4-159.

4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.

4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.

4.48 Half-precision floating-point number format on page 4-164.

4.49 Compiler support for floating-point computations and linkage on page 4-165.

4.50 Types of floating-point linkage on page 4-166.

4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.

4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.

Related information

Institute of Electrical and Electronics Engineers.

4.46 Implementation of Vector Floating-Point (VFP) support code

For convenience, an implementation of VFP support code that can be used in your system is provided with your installation of the ARM compilation tools.

The support code comprises:

- The libraries `vfpsupport.l` and `vfpsupport.b` for emulating VFP operations bounced by the hardware.

  These files are located in the `\lib\armlib` subdirectory of your installation.
- C source code and assembly language source code implementing top-level, second-level and user-level interrupt handlers.

  These files can be found in the `vfpsupport` subdirectory of the `Examples` directory of your ARM compilation tools distribution at *`install_directory`*`\Examples\...\vfpsupport.`

  These files might require modification to integrate VFP support with your operating system.
- C source code and assembly language source code for accessing subarchitecture functionality of VFP coprocessors.

  These files are located in the `vfpsupport` subdirectory of the `Examples` directory of your ARM compilation tools distribution at *`install_directory`*`\Examples\...\vfpsupport.`

When the VFP coprocessor bounces an instruction, an Undefined Instruction exception is signaled to the processor and the VFP support code is entered through the Undefined Instruction vector. The top-level and second-level interrupt handlers perform some initial processing of the signal, for example, ensuring that the exception is not caused by an illegal instruction. The user-level interrupt handler then calls the appropriate library function in the library `vfpsupport.l` or `vfpsupport.b` to emulate the VFP operation in software.

Note

You do not have to use VFP support code:
- When building with `--fpmode=std`.
- When no trapping of uncommon or exceptional cases is required.
- When the VFP coprocessor is operating in RunFast mode.
- When the hardware coprocessor is a VFPv3-based system.

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.
4.42 Default selection of hardware or software floating-point support on page 4-156.
4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.
4.44 Vector Floating-Point (VFP) architectures on page 4-159.
4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.
4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.
4.48 Half-precision floating-point number format on page 4-164.
4.49 Compiler support for floating-point computations and linkage on page 4-165.
4.50 Types of floating-point linkage on page 4-166.
4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.
4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.
7.67 --fpmode=model on page 7-341.

Related information

ARM Application Note 133 - Using VFP with RVDS.

4.47 Compiler and library support for half-precision floating-point numbers

Half-precision is a floating-point format that occupies 16 bits.

Half-precision floating-point numbers are provided by:
- The *Vector Floating-Point* (VFP) Version 4 architecture.
- An optional extension to the VFPv3 architecture.

If a VFP coprocessor is not available, or if a VFPv3 coprocessor is used that does not have the extension, half-precision floating-point numbers are supported through the floating-point library `fplib`.

Half-precision floating-point numbers can only be used when selected with the `--fp16_format=`*`format`* compiler command-line option.

The C++ name mangling for the half-precision data type is specified in the C++ generic *Application Binary Interface* (ABI).

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.
4.42 Default selection of hardware or software floating-point support on page 4-156.
4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.
4.44 Vector Floating-Point (VFP) architectures on page 4-159.
4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.
4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.
4.48 Half-precision floating-point number format on page 4-164.
4.49 Compiler support for floating-point computations and linkage on page 4-165.
4.50 Types of floating-point linkage on page 4-166.
4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.
4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.
7.66 --fp16_format=format on page 7-340.

Related information

C++ ABI for the ARM Architecture.
Floating-point Support.

4.48 Half-precision floating-point number format

The half-precision floating-point formats available are `ieee` and `alternative`. In both formats, the basic layout of the 16-bit number is the same.

The half-precision floating-point format is as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E					T									

Figure 4-1 Half-precision floating-point format

Where:

```
S (bit[15]):      Sign bit
E (bits[14:10]):  Biased exponent
T (bits[9:0]):    Mantissa.
```

The meanings of these fields depend on the format that is selected.

The IEEE half-precision format is as follows:

```
IF E==31:
   IF T==0: Value = Signed infinity
   IF T!=0: Value = Nan
             T[9] determines Quiet or Signalling:
                  0: Quiet NaN
                  1: Signalling NaN
IF 0<E<31:
   Value = (-1)^S x 2^(E-15) x (1 + (2^(-10) x T))
IF E==0:
   IF T==0: Value = Signed zero
   IF T!=0: Value = (-1)^S x 2^(-14) x (0 + (2^(-10) x T))
```

The alternative half-precision format is as follows:

```
IF 0<E<32:
   Value = (-1)^S x 2^(E-15) x (1 + (2^(-10) x T))
IF E==0:
   IF T==0: Value = Signed zero
   IF T!=0: Value = (-1)^S x 2^(-14) x (0 + (2^(-10) x T))
```

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.
4.42 Default selection of hardware or software floating-point support on page 4-156.
4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.
4.44 Vector Floating-Point (VFP) architectures on page 4-159.
4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.
4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.
4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.
4.49 Compiler support for floating-point computations and linkage on page 4-165.
4.50 Types of floating-point linkage on page 4-166.
4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.
4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.
7.66 --fp16_format=format on page 7-340.

Related information

Institute of Electrical and Electronics Engineers.

4.49 Compiler support for floating-point computations and linkage

It is important to understand the difference between floating-point computations and floating-point linkage.

Floating-point computations are performed by hardware coprocessor instructions or by library functions.

Floating-point linkage is concerned with how arguments are passed between functions that use floating-point variables.

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.
4.42 Default selection of hardware or software floating-point support on page 4-156.
4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.
4.44 Vector Floating-Point (VFP) architectures on page 4-159.
4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.
4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.
4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.
4.48 Half-precision floating-point number format on page 4-164.
4.50 Types of floating-point linkage on page 4-166.
4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.
4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.
9.45 __attribute__((pcs("calling_convention"))) function attribute on page 9-564.
9.15 __softfp on page 9-531.

4.50 Types of floating-point linkage

Different types of floating-point linkage provide different benefits.

The types of floating-point linkage are:
- Software floating-point linkage.
- Hardware floating-point linkage.

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.

Hardware floating-point linkage uses the *Vector Floating-Point* (VFP) coprocessor registers to pass the arguments and return value.

The benefit of using software floating-point linkage is that the resulting code can be run on a processor with or without a VFP coprocessor. It is not dependent on the presence of a VFP hardware coprocessor, and it can be used with or without a VFP coprocessor present.

The benefit of using hardware floating-point linkage is that it is more efficient than software floating-point linkage, but you must have a VFP coprocessor.

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.

4.42 Default selection of hardware or software floating-point support on page 4-156.

4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.

4.44 Vector Floating-Point (VFP) architectures on page 4-159.

4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.

4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.

4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.

4.48 Half-precision floating-point number format on page 4-164.

4.49 Compiler support for floating-point computations and linkage on page 4-165.

4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.

4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.

Related information

Procedure Call Standard for the ARM Architecture.

4.51 Compiler options for floating-point linkage and computations

Compiler options determine the type of floating-point linkage and floating-point computations.

By specifying the type of floating-point linkage and floating-point computations you require, you can determine, from the following table, the associated compiler command-line options that are available.

Table 4-13 Compiler options for floating-point linkage and floating-point computations

Linkage		Computations			
Hardware FP linkage	**Software FP linkage**	**Hardware FP coprocessor**	**Software FP library (fplib)**	**Compiler options**	
No	Yes	No	Yes	`--fpu=softvfp`	`--apcs=/softfp`
No	Yes	Yes	No	`--fpu=softvfp+vfpv2` `--fpu=softvfp+vfpv3` `--fpu=softvfp+vfpv3_fp16` `--fpu=softvfp+vfpv3_d16` `--fpu=softvfp+vfp3_d16_fp16` `--fpu=softvfp+vfpv4` `--fpu=softvfp+vfpv4_d16` `--fpu=softvfp+fpv4-sp`	`--apcs=/softfp`
Yes	No	Yes	No	`--fpu=vfp` `--fpu=vfpv2` `--fpu=vfpv3` `--fpu=vfpv3_fp16` `--fpu=vfpv3_dp16` `--fpu=vfpv3_d16_fp16` `--fpu=vpfv4` `--fpu=vfpv4_d16` `--fpu=fpv4-sp`	`--apcs=/hardfp`

`softvfp` specifies software floating-point linkage. When software floating-point linkage is used, either:

- The calling function and the called function must be compiled using one of the options `--softvfp`, `--fpu softvfp+vfpv2`, `--fpu softvfp+vfpv3`, `--fpu softvfp+vfpv3_fp16`, `softvfp+vfpv3_d16`, `softvfp+vfpv3_d16_fp16`, `softvfp+vfpv4`, `softvfp+vfpv4_d16`, or `softvfp+fpv4-sp`.
- The calling function and the called function must be declared using the `__softfp` keyword.

Each of the options `--fpu softvfp`, `--fpu softvfp+vfpv2`,`--fpu softvfp+vfpv3`, `--fpu softvfp+vfpv3_fp16`, `--fpu softvfpv3_d16`, `--fpu softvfpv3_d16_fp16`, `--fpu softvfp+vfpv4`, `softvfp+vfpv4_d16` and `softvfp+fpv4-sp` specify software floating-point linkage across the whole file. In contrast, the `__softfp` keyword enables software floating-point linkage to be specified on a function by function basis.

Note

Rather than having separate compiler options to select the type of floating-point linkage you require and the type of floating-point computations you require, you use one compiler option, `--fpu`, to select both.

For example, `--fpu=softvfp+vfpv2` selects *software* floating-point linkage, and a *hardware* coprocessor for the computations. Whenever you use `softvfp`, you are specifying software floating-point linkage.

If you use the `--fpu` option, you must know the VFP architecture version implemented in the target processor. An alternative to `--fpu=softvfp+...` is `--apcs=/softfp`. This gives software linkage with whatever VFP architecture version is implied by `--cpu`. `--apcs=/softfp` and `--apcs=/hardfp` are alternative ways of requesting the integer or floating-point variant of the *Procedure Call Standard for the ARM Architecture* (AAPCS).

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.
4.42 Default selection of hardware or software floating-point support on page 4-156.
4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.
4.44 Vector Floating-Point (VFP) architectures on page 4-159.
4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.
4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.
4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.
4.48 Half-precision floating-point number format on page 4-164.
4.49 Compiler support for floating-point computations and linkage on page 4-165.
4.50 Types of floating-point linkage on page 4-166.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.
4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.
7.6 --apcs=qualifier...qualifier on page 7-273.
7.69 --fpu=name compiler option on page 7-344.
7.93 --library_interface=lib on page 7-370.
9.15 __softfp on page 9-531.
9.98 #pragma softfp_linkage, #pragma no_softfp_linkage on page 9-619.

Related information

Procedure Call Standard for the ARM Architecture.

4.52 Floating-point linkage and computational requirements of compiler options

There are various valid combinations of FPU options and processors.

The following table sets out the FPU options, and their capabilities and requirements.

Table 4-14 FPU-option capabilities and requirements

FPU name	Hardware FP linkage	d0-d15 registers	d16-d31 registers	VFP instructions	Half precision	Single precision	Double precision
`softvfp`	No	No	No	No	No	No	No
`softvfp+vfpv2`	No	Yes	No	Yes	No	Yes	Yes
`softvfp+vfpv3`	No	Yes	Yes	Yes	No	Yes	Yes
`softvfp +vfpv3_fp16`	No	Yes	Yes	Yes	Yes	Yes	Yes
`softvfp+vfpv3_d16`	No	Yes	No	Yes	No	Yes	Yes
`softvfp +vfpv3_d16_fp16`	No	Yes	No	Yes	Yes	Yes	Yes
`softvfp +vfpv3_sp_d16`	No	Yes	No	Yes	Yes	Yes	No
`softvfp+vfpv4`	No	Yes	Yes	Yes	Yes	Yes	Yes
`softvfp+vfpv4_d16`	No	Yes	No	Yes	Yes	Yes	Yes
`softvfp +vfpv4_sp_d16`	No	Yes	No	Yes	Yes	Yes	No
`softvfp+fpv4-sp`	No	Yes	No	Yes	Yes	Yes	No
`vfp`	Yes	Yes	No	Yes	No	Yes	Yes
`vfpv2`	Yes	Yes	No	Yes	No	Yes	Yes
`vfpv3`	Yes	Yes	Yes	Yes	No	Yes	Yes
`vfpv3_fp16`	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`vfpv3_d16`	Yes	Yes	No	Yes	No	Yes	Yes
`vfpv3_d16_fp16`	Yes	Yes	No	Yes	Yes	Yes	Yes
`vfpv3_sp_d16`	Yes	Yes	No	Yes	Yes	Yes	No
`vfpv4`	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`vfpv4_d16`	Yes	Yes	No	Yes	Yes	Yes	Yes
`vfpv4_sp_d16`	Yes	Yes	No	Yes	Yes	Yes	No
`fpv4-sp`	Yes	Yes	No	Yes	Yes	Yes	No

Note

You can specify the floating-point linkage, independently of the VFP architecture, with `--apcs`.

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.

4.42 Default selection of hardware or software floating-point support on page 4-156.
4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.
4.44 Vector Floating-Point (VFP) architectures on page 4-159.
4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.
4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.
4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.
4.48 Half-precision floating-point number format on page 4-164.
4.49 Compiler support for floating-point computations and linkage on page 4-165.
4.50 Types of floating-point linkage on page 4-166.
4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.
7.6 --apcs=qualifier...qualifier on page 7-273.
7.69 --fpu=name compiler option on page 7-344.

4.53 Processors and their implicit Floating-Point Units (FPUs)

Not every ARM processor has an FPU, but every one has an implicit `--fpu` option.

The following table lists the implicit `--fpu` option for each processor `--cpu` option.

Table 4-15 Implicit FPUs of processors

Processor name	**FPU** name
ARM processors designed by ARM Limited	
`ARM7EJ-S`	`SoftVFP`
`ARM7TDMI`	`SoftVFP`
`ARM7TDMI-S`	`SoftVFP`
`ARM720T`	`SoftVFP`
`ARM9E-S`	`SoftVFP`
`ARM9TDMI`	`SoftVFP`
`ARM920T`	`SoftVFP`
`ARM922T`	`SoftVFP`
`ARM926EJ-S`	`SoftVFP`
`ARM946E-S`	`SoftVFP`
`ARM966E-S`	`SoftVFP`
`Cortex-M0`	`SoftVFP`
`Cortex-M0plus`	`SoftVFP`
`Cortex-M1`	`SoftVFP`
`Cortex-M1.os_extension`	`SoftVFP`
`Cortex-M1.no_os_extension`	`SoftVFP`
`Cortex-M3`	`SoftVFP`
`Cortex-M3-rev0`	`SoftVFP`
`Cortex-M4`	`SoftVFP`
`Cortex-M4.fp.sp`	`FPv4-SP`
`Cortex-M7`	`SoftVFP`
`Cortex-M7.fp.sp`	`FPv5-SP`
`Cortex-M7.fp.dp`	`FPv5_D16`
`Cortex-R4`	`SoftVFP`
`Cortex-R4F`	`VFPv3_D16`
`Cortex-R5`	`SoftVFP`
`Cortex-R5-rev1`	`SoftVFP`
`Cortex-R5F`	`VFPv3_D16`
`Cortex-R5F-rev1`	`VFPv3_D16`
`Cortex-R5F-rev1.sp`	`VFPv3_SP_D16`

Table 4-15 Implicit FPUs of processors (continued)

Processor name	FPU name
`Cortex-R7`	`VFPv3_D16_FP16`
`Cortex-R7.no_vfp`	`SoftVFP`
`SC000`	`SoftVFP`
`SC300`	`SoftVFP`
ARM processors designed by ARM licensees	

Note

You can:
- Specify a different FPU with `--fpu`.
- Specify the floating-point linkage, independently of the FPU architecture, with `--apcs`.
- Display the complete expanded command line, including the FPU, with `--echo`.

Related concepts

4.41 Compiler support for floating-point arithmetic on page 4-154.
4.42 Default selection of hardware or software floating-point support on page 4-156.
4.43 Example of hardware and software support differences for floating-point arithmetic on page 4-157.
4.44 Vector Floating-Point (VFP) architectures on page 4-159.
4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.
4.46 Implementation of Vector Floating-Point (VFP) support code on page 4-161.
4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.
4.48 Half-precision floating-point number format on page 4-164.
4.49 Compiler support for floating-point computations and linkage on page 4-165.
4.50 Types of floating-point linkage on page 4-166.
4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

4.52 Floating-point linkage and computational requirements of compiler options on page 4-169.
7.6 --apcs=qualifier...qualifier on page 7-273.
7.54 --echo on page 7-328.
7.69 --fpu=name compiler option on page 7-344.

4.54 Integer division-by-zero errors in C code

For targets that do not support hardware division instructions (for example `SDIV` and `UDIV`), you can trap and identify integer division-by-zero errors with the appropriate C library helper functions, `__aeabi_idiv0()` and `__rt_raise()`.

Trapping integer division-by-zero errors with __aeabi_idiv0()

You can trap integer division-by-zero errors with the C library helper function `__aeabi_idiv0()` so that division by zero returns some standard result, for example zero.

Integer division is implemented in code through the C library helper functions `__aeabi_idiv()` and `__aeabi_uidiv()`. Both functions check for division by zero.

When integer division by zero is detected, a branch to `__aeabi_idiv0()` is made. To trap the division by zero, therefore, you only have to place a breakpoint on `__aeabi_idiv0()`.

The library provides two implementations of `__aeabi_idiv0()`. The default one does nothing, so if division by zero is detected, the division function returns zero. However, if you use signal handling, an alternative implementation is selected that calls `__rt_raise(SIGFPE, DIVBYZERO)`.

If you provide your own version of `__aeabi_idiv0()`, then the division functions call this function. The function prototype for `__aeabi_idiv0()` is:

```
int __aeabi_idiv0(void);
```

If `__aeabi_idiv0()` returns a value, that value is used as the quotient returned by the division function.

On entry into `__aeabi_idiv0()`, the link register `LR` contains the address of the instruction *after* the call to the `__aeabi_uidiv()` division routine in your application code.

The offending line in the source code can be identified by looking up the line of C code in the debugger at the address given by `LR`.

If you want to examine parameters and save them for postmortem debugging when trapping `__aeabi_idiv0`, you can use the `$Super$$` and `$Sub$$` mechanism:

1. Prefix `__aeabi_idiv0()` with `$Super$$` to identify the original unpatched function `__aeabi_idiv0()`.
2. Use `__aeabi_idiv0()` prefixed with `$Super$$` to call the original function directly.
3. Prefix `__aeabi_idiv0()` with `$Sub$$` to identify the new function to be called in place of the original version of `__aeabi_idiv0()`.
4. Use `__aeabi_idiv0()` prefixed with `$Sub$$` to add processing before or after the original function `__aeabi_idiv0()`.

The following example shows how to intercept `__aeabi_div0` using the `$Super$$` and `$Sub$$` mechanism.

```
extern void $Super$$__aeabi_idiv0(void);
/* this function is called instead of the original __aeabi_idiv0() */
void $Sub$$__aeabi_idiv0()
{
    // insert code to process a divide by zero
    ...
    // call the original __aeabi_idiv0 function
    $Super$$__aeabi_idiv0();
}
```

Trapping integer division-by-zero errors with __rt_raise()

By default, integer division by zero returns zero. If you want to intercept division by zero, you can re-implement the C library helper function `__rt_raise()`.

The function prototype for `__rt_raise()` is:

```
void __rt_raise(int signal, int type);
```

If you re-implement `__rt_raise()`, then the library automatically provides the signal-handling library version of `__aeabi_idiv0()`, which calls `__rt_raise()`, then that library version of `__aeabi_idiv0()` is included in the final image.

In that case, when a divide-by-zero error occurs, `__aeabi_idiv0()` calls `__rt_raise(SIGFPE, DIVBYZERO)`. Therefore, if you re-implement `__rt_raise()`, you must check `(signal == SIGFPE) && (type == DIVBYZERO)` to determine if division by zero has occurred.

Related information

Run-time ABI for the ARM Architecture.

4.55 Software floating-point division-by-zero errors in C code

Floating-point division-by-zero errors in software can be trapped and identified using a combination of intrinsics and C library helper functions.

Specifically:

- The `__ieee_status` intrinsic lets you trap floating-point division-by-zero errors.
- Placing a breakpoint on `_fp_trapveneer()` lets you identify software floating-point division-by-zero errors.
- Intercepting `_fp_trapveneer()` using the `$Super$$` and `$Sub$$` mechanism lets you save parameters for debugging.

Related concepts

4.56 About trapping software floating-point division-by-zero errors on page 4-176.

4.57 Identification of software floating-point division-by-zero errors on page 4-177.

4.58 Software floating-point division-by-zero debugging on page 4-179.

4.56 About trapping software floating-point division-by-zero errors

Software floating-point division-by-zero errors can be trapped with the `__ieee_status` intrinsic.

```
__ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_DIVBYZERO);
```

This traps any division-by-zero errors in code, and untraps all other exceptions, as illustrated in the following example:

```
#include <stdio.h>
#include <fenv.h>
int main(void)
{    float a, b, c;
    // Trap the Invalid Operation exception and untrap all other
    // exceptions:
    __ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_DIVBYZERO);
    c = 0;
    a = b / c;
    printf("b / c = %f, ", a); // trap division-by-zero error
    return 0;
}
```

Related concepts

4.55 Software floating-point division-by-zero errors in C code on page 4-175.

4.57 Identification of software floating-point division-by-zero errors on page 4-177.

4.58 Software floating-point division-by-zero debugging on page 4-179.

Related information

__ieee_status().

4.57 Identification of software floating-point division-by-zero errors

You can use the C library helper function `_fp_trapveneer()` to identify the location of a software floating-point division-by-zero error.

`_fp_trapveneer()` is called whenever an exception occurs. On entry into this function, the state of the registers is unchanged from when the exception occurred. Therefore, to find the address of the function in the application code that contains the arithmetic operation that resulted in the exception, a breakpoint can be placed on the function `_fp_trapveneer()` and `LR` can be inspected.

For example, consider the following example C code:

```
#include <stdio.h>
#include <fenv.h>
int main(void)
{    float a, b, c;
    // Trap the Invalid Operation exception and untrap all other
    // exceptions:
    __ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_DIVBYZERO);
    c = 0;
    b = 5.366789;
    a = b / c;
    printf("b / c = %f, ", a); // trap division-by-zero error
    return 0;
}
```

This example code is compiled with the following command:

```
armcc --fpmode ieee_full
```

The compiled example disassembles to the following code:

```
main:
    0x000080E0 : PUSH     {r4,lr}
    0x000080E4 : MOV      r1,#0x200
    0x000080E8 : MOV      r0,#0x9f00
    0x000080EC : BL       __ieee_status ; 0xB9B8
    0x000080F0 : MOV      r4,#0
    0x000080F4 : LDR      r0,[pc,#40] ; [0x8124] = 0x891E2153
    0x000080F8 : LDR      r1,[pc,#40] ; [0x8128] = 0x40157797
    0x000080FC : BL       __aeabi_d2f ; 0xA948
    0x00008100 : MOV      r1,r4
    0x00008104 : BL       __aeabi_fdiv ; 0xB410
    0x00008108 : BL       __aeabi_f2d ; 0xB388
    0x0000810C : MOV      r2,r0
    0x00008110 : MOV      r3,r1
    0x00008114 : ADR      r0,{pc}+0x18 ; 0x812c
    0x00008118 : BL       __2printf ; 0x813C
    0x0000811C : MOV      r0,#0
    0x00008120 : POP      {r4,pc}
    0x00008124 : DCD      0x891E2153
    0x00008128 : DCD      0x40157797
    0x0000812C : DCD      0x202F2062
    0x00008130 : DCD      0x203D2063
    0x00008134 : DCD      0x202C6625
    0x00008138 : DCD      0x00000000
```

Placing a breakpoint on `_fp_trapveneer()` and executing the disassembly in the debug monitor produces:

```
> run
Execution stopped at breakpoint 1: S:0x0000BAC8
In _fp_trapveneer (no debug info)
S:0x0000BAC8   PUSH     {r12,lr}
```

Then, inspection of the registers shows:

```
  r0: 0x40ABBCBC     r1: 0x00000000     r2: 0x00000000     r3: 0x00000000
  r4: 0x0000C1DC     r5: 0x0000BD44     r6: 0x00000000     r7: 0x00000000
  r8: 0x00000000     r9: 0x00000000    r10: 0x0000BC1C    r11: 0x00000000
 r12: 0x08000004     SP: 0x0FFFFFF8     LR: 0x00008108     PC: 0x0000BAC8
CPSR: 0x000001D3
```

The address contained in the link register `LR` is set to 0x8108, the address of the instruction after the instruction `BL __aeabi_fdiv` that resulted in the exception.

Related concepts

4.55 Software floating-point division-by-zero errors in C code on page 4-175.
4.56 About trapping software floating-point division-by-zero errors on page 4-176.
4.58 Software floating-point division-by-zero debugging on page 4-179.

4.58 Software floating-point division-by-zero debugging

Parameters for postmortem debugging can be saved by intercepting `_fp_trapveneer()`.

You can use the `$Super$$` and `$Sub$$` mechanism to intervene in all calls to `_fp_trapveneer()`.

For example:

```
    AREA foo, CODE
IMPORT |$Super$$_fp_trapveneer|
EXPORT |$Sub$$_fp_trapveneer|
    |$Sub$$_fp_trapveneer|
;; Add code to save whatever registers you require here
;; Take care not to corrupt any needed registers
    B |$Super$$_fp_trapveneer|
    END
```

Related concepts

4.55 Software floating-point division-by-zero errors in C code on page 4-175.

4.56 About trapping software floating-point division-by-zero errors on page 4-176.

4.57 Identification of software floating-point division-by-zero errors on page 4-177.

Related information

Use of $Super$$ and $Sub$$ to patch symbol definitions.

4.59 New language features of C99

The 1999 C99 standard introduces several new language features.

These new features include:

- Some features similar to extensions to C90 offered in the GNU compiler, for example, macros with a variable number of arguments.

  Note

  The implementations of extensions to C90 in the GNU compiler are not always compatible with the implementations of similar features in C99.

- Some features available in C++, such as `//` comments and the ability to mix declarations and statements.
- Some entirely new features, for example complex numbers, restricted pointers and designated initializers.
- New keywords and identifiers.
- Extended syntax for the existing C90 language.

A selection of new features in C99 that might be of interest to developers using them for the first time are documented.

Note

C90 is compatible with Standard C++ in the sense that the language specified by the standard is a subset of C++, except for a few special cases. New features in the C99 standard mean that C99 is no longer compatible with C++ in this sense.

Some examples of special cases where the language specified by the C90 standard is not a subset of C++ include support for `//` comments and merging of the typedef and structure tag namespaces. For example, in C90 the following code expands to `x = a / b - c;` because `/* hello world */` is deleted, but in C ++ and C99 it expands to `x = a - c;` because everything from `//` to the end of the first line is deleted:

```
x = a //* hello world */ b
    - c;
```

The following code demonstrates how typedef and the structure tag are treated differently between C (90 and 99) and C++ because of their merged namespaces:

```
typedef int a;
{
  struct a { int x, y; };
  printf("%d\n", sizeof(a));
}
```

In C 90 and C99, this code defines two types with separate names whereby `a` is a typedef for `int` and `struct a` is a structure type containing two integer data types. `sizeof(a)` evaluates to `sizeof(int)`.

In C++, a structure type can be addressed using only its tag. This means that when the definition of `struct a` is in scope, the name `a` used on its own refers to the structure type rather than the typedef, so in C++ `sizeof(a)` is greater than `sizeof(int)`.

Related concepts

4.61 // comments in C99 and C90 on page 4-183.
4.62 Compound literals in C99 on page 4-184.
4.63 Designated initializers in C99 on page 4-185.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.65 Flexible array members in C99 on page 4-187.
4.66 __func__ predefined identifier in C99 on page 4-188.

4.67 inline functions in C99 on page 4-189.

4.68 long long data type in C99 and C90 on page 4-190.

4.69 Macros with a variable number of arguments in C99 on page 4-191.

4.70 Mixed declarations and statements in C99 on page 4-192.

4.71 New block scopes for selection and iteration statements in C99 on page 4-193.

4.72 _Pragma preprocessing operator in C99 on page 4-194.

4.73 Restricted pointers in C99 on page 4-195.

4.75 Complex numbers in C99 on page 4-197.

4.60 New library features of C99

The C99 standard introduces several new library features of interest to programmers.

These new features include:

- Some features similar to extensions to the C90 standard libraries offered in UNIX standard libraries, for example, the `snprintf` family of functions.
- Some entirely new library features, for example, the standardized floating-point environment offered in `<fenv.h>`.
- New libraries, and new macros and functions for existing C90 libraries.

A selection of new features in C99 that might be of interest to developers using them for the first time are documented.

Note

C90 is compatible with Standard C++ in the sense that the language specified by the standard is a subset of C++, except for a few special cases. New features in the C99 standard mean that C99 is no longer compatible with C++ in this sense.

Many library features that are new to C99 are available in C90 and C++. Some require macros such as `USE_C99_ALL` or `USE_C99_MATH` to be defined before the `#include`.

Related concepts

4.74 Additional <math.h> library functions in C99 on page 4-196.
4.75 Complex numbers in C99 on page 4-197.
4.76 Boolean type and <stdbool.h> in C99 on page 4-198.
4.77 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99 on page 4-199.
4.78 <fenv.h> floating-point environment access in C99 on page 4-200.
4.79 <stdio.h> snprintf family of functions in C99 on page 4-201.
4.80 <tgmath.h> type-generic math macros in C99 on page 4-202.
4.81 <wchar.h> wide character I/O functions in C99 on page 4-203.

4.61 // comments in C99 and C90

In C99 you can use `//` to indicate the start of a one-line comment, like in C++. In C90 mode you can use `//` comments providing you do not specify `--strict`.

Related concepts

4.59 New language features of C99 on page 4-180.
4.62 Compound literals in C99 on page 4-184.
4.63 Designated initializers in C99 on page 4-185.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.65 Flexible array members in C99 on page 4-187.
4.66 __func__ predefined identifier in C99 on page 4-188.
4.67 inline functions in C99 on page 4-189.
4.68 long long data type in C99 and C90 on page 4-190.
4.69 Macros with a variable number of arguments in C99 on page 4-191.
4.70 Mixed declarations and statements in C99 on page 4-192.
4.71 New block scopes for selection and iteration statements in C99 on page 4-193.
4.72 _Pragma preprocessing operator in C99 on page 4-194.
4.73 Restricted pointers in C99 on page 4-195.
4.75 Complex numbers in C99 on page 4-197.

Related references

7.156 --strict, --no_strict on page 7-439.
8.7 // comments on page 8-472.

4.62 Compound literals in C99

ISO C99 supports compound literals. A compound literal looks like a cast followed by an initializer.

Its value is an object of the type specified in the cast, containing the elements specified in the initializer. It is an lvalue.

For example:

```
int *y = (int []) {1, 2, 3};
int *z = (int [3]) {1};
```

Note

`int *y = (int []) {1, 2, 3};` is accepted by the compiler, but `int y[] = (int []) {1, 2, 3};` is not accepted as a high-level (global) initialization.

In the following example source code, the compound literals are:

- `(struct T) { 43, "world"}`
- `&(struct T) {.b = "hello", .a = 47}`
- `&(struct T) {43, "hello"}`
- `(int[]){1, 2, 3}`

```
struct T
{
   int a;
   char *b;
} t2;
void g(const struct T *t);
void f()
{
   int x[10];
   ...
   t2 = (struct T) {43, "world"};
   g(&(struct T) {.b = "hello", .a = 47});
   g(&(struct T) {43, "bye"});
   memcpy(x, (int[]){1, 2, 3}, 3 * sizeof(int));
}
```

Related concepts

4.59 New language features of C99 on page 4-180.
4.61 // comments in C99 and C90 on page 4-183.
4.63 Designated initializers in C99 on page 4-185.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.65 Flexible array members in C99 on page 4-187.
4.66 __func__ predefined identifier in C99 on page 4-188.
4.67 inline functions in C99 on page 4-189.
4.68 long long data type in C99 and C90 on page 4-190.
4.69 Macros with a variable number of arguments in C99 on page 4-191.
4.70 Mixed declarations and statements in C99 on page 4-192.
4.71 New block scopes for selection and iteration statements in C99 on page 4-193.
4.72 _Pragma preprocessing operator in C99 on page 4-194.
4.73 Restricted pointers in C99 on page 4-195.
4.75 Complex numbers in C99 on page 4-197.

4.63 Designated initializers in C99

In C90, there is no way to initialize specific members of arrays, structures, or unions. C99 supports the initialization of specific members of an array, structure, or union by either name or subscript through the use of designated initializers.

For example:

```
typedef struct
{
    char *name;
    int rank;
} data;
data vars[10] = { [0].name = "foo", [0].rank = 1,
                  [1].name = "bar", [1].rank = 2,
                  [2].name = "baz",
                  [3].name = "gazonk" };
```

Members of an aggregate that are not explicitly initialized are initialized to zero by default.

Related concepts

4.59 New language features of C99 on page 4-180.
4.61 // comments in C99 and C90 on page 4-183.
4.62 Compound literals in C99 on page 4-184.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.65 Flexible array members in C99 on page 4-187.
4.66 __func__ predefined identifier in C99 on page 4-188.
4.67 inline functions in C99 on page 4-189.
4.68 long long data type in C99 and C90 on page 4-190.
4.69 Macros with a variable number of arguments in C99 on page 4-191.
4.70 Mixed declarations and statements in C99 on page 4-192.
4.71 New block scopes for selection and iteration statements in C99 on page 4-193.
4.72 _Pragma preprocessing operator in C99 on page 4-194.
4.73 Restricted pointers in C99 on page 4-195.
4.75 Complex numbers in C99 on page 4-197.

4.64 Hexadecimal floating-point numbers in C99

C99 supports floating-point numbers that can be written in hexadecimal format.

For example:

```
float hex_floats(void)
{
    return 0x1.fp3; // 1 15/16 * 2^3
}
```

In hexadecimal format the exponent is a decimal number that indicates the power of two by which the significant part is multiplied. Therefore 0x1.fp3 = 1.9375 * 8 = 1.55e1.

C99 also adds `%a` and `%A` format for `printf()`.

Related concepts

4.59 New language features of C99 on page 4-180.

4.61 // comments in C99 and C90 on page 4-183.

4.62 Compound literals in C99 on page 4-184.

4.63 Designated initializers in C99 on page 4-185.

4.65 Flexible array members in C99 on page 4-187.

4.66 __func__ predefined identifier in C99 on page 4-188.

4.67 inline functions in C99 on page 4-189.

4.68 long long data type in C99 and C90 on page 4-190.

4.69 Macros with a variable number of arguments in C99 on page 4-191.

4.70 Mixed declarations and statements in C99 on page 4-192.

4.71 New block scopes for selection and iteration statements in C99 on page 4-193.

4.72 _Pragma preprocessing operator in C99 on page 4-194.

4.73 Restricted pointers in C99 on page 4-195.

4.75 Complex numbers in C99 on page 4-197.

4.65 Flexible array members in C99

In a **`struct`** with more than one member, the last member of the **`struct`** can have incomplete array type. Such a member is called a *flexible array member* of the **`struct`**.

——— **Note** ———

When a **`struct`** has a flexible array member, the entire **`struct`** itself has incomplete type.

———————————

Flexible array members enable you to mimic dynamic type specification in C in the sense that you can defer the specification of the array size to runtime. For example:

```
extern const int n;
typedef struct
{
    int len;
    char p[];
} str;
void foo(void)
{
    size_t str_size = sizeof(str);  // equivalent to offsetoff(str, p)
    str *s = malloc(str_size + (sizeof(char) * n));
}
```

Related concepts

4.59 New language features of C99 on page 4-180.
4.61 // comments in C99 and C90 on page 4-183.
4.62 Compound literals in C99 on page 4-184.
4.63 Designated initializers in C99 on page 4-185.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.66 __func__ predefined identifier in C99 on page 4-188.
4.67 inline functions in C99 on page 4-189.
4.68 long long data type in C99 and C90 on page 4-190.
4.69 Macros with a variable number of arguments in C99 on page 4-191.
4.70 Mixed declarations and statements in C99 on page 4-192.
4.71 New block scopes for selection and iteration statements in C99 on page 4-193.
4.72 _Pragma preprocessing operator in C99 on page 4-194.
4.73 Restricted pointers in C99 on page 4-195.
4.75 Complex numbers in C99 on page 4-197.

4.66 __func__ predefined identifier in C99

The `__func__` predefined identifier provides a means of obtaining the name of the current function.

For example, the function:

```
void foo(void)
{
    printf("This function is called '%s'.\n", __func__);
}
```

prints:

```
This function is called 'foo'.
```

Related concepts

4.59 New language features of C99 on page 4-180.
4.61 // comments in C99 and C90 on page 4-183.
4.62 Compound literals in C99 on page 4-184.
4.63 Designated initializers in C99 on page 4-185.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.65 Flexible array members in C99 on page 4-187.
4.67 inline functions in C99 on page 4-189.
4.68 long long data type in C99 and C90 on page 4-190.
4.69 Macros with a variable number of arguments in C99 on page 4-191.
4.70 Mixed declarations and statements in C99 on page 4-192.
4.71 New block scopes for selection and iteration statements in C99 on page 4-193.
4.72 _Pragma preprocessing operator in C99 on page 4-194.
4.73 Restricted pointers in C99 on page 4-195.
4.75 Complex numbers in C99 on page 4-197.

Related references

9.159 Built-in function name variables on page 9-703.

4.67 inline functions in C99

The C99 keyword **`inline`** hints to the compiler that invocations of a function qualified with **`inline`** are to be expanded inline.

For example:

```
inline int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

The compiler inlines a function qualified with **`inline`** only if it is reasonable to do so. It is free to ignore the hint if inlining the function adversely affects performance.

Note

The **`__inline`** keyword is available in C90.

Note

The semantics of **`inline`** in C99 are different to the semantics of **`inline`** in Standard C++.

Related concepts

4.59 New language features of C99 on page 4-180.
4.61 // comments in C99 and C90 on page 4-183.
4.62 Compound literals in C99 on page 4-184.
4.63 Designated initializers in C99 on page 4-185.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.65 Flexible array members in C99 on page 4-187.
4.66 __func__ predefined identifier in C99 on page 4-188.
4.68 long long data type in C99 and C90 on page 4-190.
4.69 Macros with a variable number of arguments in C99 on page 4-191.
4.70 Mixed declarations and statements in C99 on page 4-192.
4.71 New block scopes for selection and iteration statements in C99 on page 4-193.
4.72 _Pragma preprocessing operator in C99 on page 4-194.
4.73 Restricted pointers in C99 on page 4-195.
4.75 Complex numbers in C99 on page 4-197.
4.20 Inline functions on page 4-131.

4.68 long long data type in C99 and C90

C99 supports the integral data type **`long long`**.

This type is 64 bits wide in the ARM compilation tools.

For example:

```
long long int j = 25902068371200;                // length of light
                                                 // day, meters
unsigned long long int i = 94607304725808000ULL; // length of light
                                                 // year, meters
```

`long long` is also available in C90 when not using `--strict`.

`__int64` is a synonym for **`long long`**. `__int64` is always available.

Related concepts

4.59 New language features of C99 on page 4-180.
4.61 // comments in C99 and C90 on page 4-183.
4.62 Compound literals in C99 on page 4-184.
4.63 Designated initializers in C99 on page 4-185.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.65 Flexible array members in C99 on page 4-187.
4.66 __func__ predefined identifier in C99 on page 4-188.
4.67 inline functions in C99 on page 4-189.
4.69 Macros with a variable number of arguments in C99 on page 4-191.
4.70 Mixed declarations and statements in C99 on page 4-192.
4.71 New block scopes for selection and iteration statements in C99 on page 4-193.
4.72 _Pragma preprocessing operator in C99 on page 4-194.
4.73 Restricted pointers in C99 on page 4-195.
4.75 Complex numbers in C99 on page 4-197.

Related references

8.12 long long on page 8-477.

4.69 Macros with a variable number of arguments in C99

You can declare a macro in C99 that accepts a variable number of arguments.

The syntax for defining such a macro is similar to that of a function. For example:

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
void Variadic_Macros_0()
{
    debug ("a test string is printed out along with %x %x %x\n", 12, 14, 20);
}
```

Related concepts

4.59 New language features of C99 on page 4-180.
4.61 // comments in C99 and C90 on page 4-183.
4.62 Compound literals in C99 on page 4-184.
4.63 Designated initializers in C99 on page 4-185.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.65 Flexible array members in C99 on page 4-187.
4.66 __func__ predefined identifier in C99 on page 4-188.
4.67 inline functions in C99 on page 4-189.
4.68 long long data type in C99 and C90 on page 4-190.
4.70 Mixed declarations and statements in C99 on page 4-192.
4.71 New block scopes for selection and iteration statements in C99 on page 4-193.
4.72 _Pragma preprocessing operator in C99 on page 4-194.
4.73 Restricted pointers in C99 on page 4-195.
4.75 Complex numbers in C99 on page 4-197.

4.70 Mixed declarations and statements in C99

C99 enables you to mix declarations and statements within compound statements, like in C++.

For example:

```
void foo(float i)
{
    i = (i > 0) ? -i : i;
    float j = sqrt(i);    // illegal in C90
}
```

Related concepts

4.59 New language features of C99 on page 4-180.

4.61 // comments in C99 and C90 on page 4-183.

4.62 Compound literals in C99 on page 4-184.

4.63 Designated initializers in C99 on page 4-185.

4.64 Hexadecimal floating-point numbers in C99 on page 4-186.

4.65 Flexible array members in C99 on page 4-187.

4.66 __func__ predefined identifier in C99 on page 4-188.

4.67 inline functions in C99 on page 4-189.

4.68 long long data type in C99 and C90 on page 4-190.

4.69 Macros with a variable number of arguments in C99 on page 4-191.

4.71 New block scopes for selection and iteration statements in C99 on page 4-193.

4.72 _Pragma preprocessing operator in C99 on page 4-194.

4.73 Restricted pointers in C99 on page 4-195.

4.75 Complex numbers in C99 on page 4-197.

4.71 New block scopes for selection and iteration statements in C99

In a **for** loop, the first expression can be a declaration, like in C++. The scope of the declaration extends to the body of the loop only.

For example:

```
extern int max;
for (int n = max - 1; n >= 0; n--)
{
    // body of loop
}
```

is equivalent to:

```
extern int max;
{
    int n = max - 1;
    for (; n >= 0; n--)
    {
        // body of loop
    }
}
```

Note

Unlike in C++, you cannot introduce new declarations in a **for**-test, **if**-test or **switch**-expression.

Related concepts

4.59 New language features of C99 on page 4-180.
4.61 // comments in C99 and C90 on page 4-183.
4.62 Compound literals in C99 on page 4-184.
4.63 Designated initializers in C99 on page 4-185.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.65 Flexible array members in C99 on page 4-187.
4.66 __func__ predefined identifier in C99 on page 4-188.
4.67 inline functions in C99 on page 4-189.
4.68 long long data type in C99 and C90 on page 4-190.
4.69 Macros with a variable number of arguments in C99 on page 4-191.
4.70 Mixed declarations and statements in C99 on page 4-192.
4.72 _Pragma preprocessing operator in C99 on page 4-194.
4.73 Restricted pointers in C99 on page 4-195.
4.75 Complex numbers in C99 on page 4-197.

4.72 _Pragma preprocessing operator in C99

C90 does not permit a `#pragma` directive to be produced as the result of a macro expansion. However, the C99 `_Pragma` operator enables you to embed a preprocessor macro in a pragma directive.

`_Pragma` is permitted in C90 if `--strict` is not specified.

For example:

```
# define RWDATA(X) PRAGMA(arm section rwdata=#X)
# define PRAGMA(X) _Pragma(#X)
RWDATA(foo)  // same as #pragma arm section rwdata="foo"
int y = 1;   // y is placed in section "foo"
```

Related concepts

4.59 New language features of C99 on page 4-180.

4.61 // comments in C99 and C90 on page 4-183.

4.62 Compound literals in C99 on page 4-184.

4.63 Designated initializers in C99 on page 4-185.

4.64 Hexadecimal floating-point numbers in C99 on page 4-186.

4.65 Flexible array members in C99 on page 4-187.

4.66 __func__ predefined identifier in C99 on page 4-188.

4.67 inline functions in C99 on page 4-189.

4.68 long long data type in C99 and C90 on page 4-190.

4.69 Macros with a variable number of arguments in C99 on page 4-191.

4.70 Mixed declarations and statements in C99 on page 4-192.

4.71 New block scopes for selection and iteration statements in C99 on page 4-193.

4.73 Restricted pointers in C99 on page 4-195.

4.75 Complex numbers in C99 on page 4-197.

4.73 Restricted pointers in C99

The C99 keyword **`restrict`** is an indication to the compiler that different object pointer types and function parameter arrays do not point to overlapping regions of memory.

This enables the compiler to perform optimizations that might otherwise be prevented because of possible aliasing.

In the following example, pointer `a` does not, and must not, point to the same region of memory as pointer `b`:

```
void copy_array(int n, int *restrict a, int *restrict b)
{
    while (n-- > 0)
        *a++ = *b++;
}
void test(void)
{
    extern int array[100];
    copy_array(50, array + 50, array);    // valid
    copy_array(50, array + 1, array);     // undefined behavior
}
```

Pointers qualified with **`restrict`** can however point to different arrays, or to different regions within an array.

It is your responsibility to ensure that **`restrict`**-qualified pointers do not point to overlapping regions of memory.

`__restrict`, permitted in C90 and C++, is a synonym for **`restrict`**.

`--restrict` enables **`restrict`** to be used in C90 and C++.

Related concepts

4.59 New language features of C99 on page 4-180.
4.61 // comments in C99 and C90 on page 4-183.
4.62 Compound literals in C99 on page 4-184.
4.63 Designated initializers in C99 on page 4-185.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.65 Flexible array members in C99 on page 4-187.
4.66 __func__ predefined identifier in C99 on page 4-188.
4.67 inline functions in C99 on page 4-189.
4.68 long long data type in C99 and C90 on page 4-190.
4.69 Macros with a variable number of arguments in C99 on page 4-191.
4.70 Mixed declarations and statements in C99 on page 4-192.
4.71 New block scopes for selection and iteration statements in C99 on page 4-193.
4.72 _Pragma preprocessing operator in C99 on page 4-194.
4.75 Complex numbers in C99 on page 4-197.

Related references

7.145 --restrict, --no_restrict on page 7-427.

4.74 Additional <math.h> library functions in C99

C99 supports additional macros, types, and functions in the standard header `<math.h>` that are not found in the corresponding C90 standard header.

New macros found in C99 that are not found in C90 include:

```
INFINITY // positive infinity
NAN      // IEEE not-a-number
```

New generic function macros found in C99 that are not found in C90 include:

```
#define isinf(x) // non-zero only if x is positive or negative infinity
#define isnan(x) // non-zero only if x is NaN
#define isless(x, y) // 1 only if x < y and x and y are not NaN, and 0 otherwise
#define isunordered(x, y) // 1 only if either x or y is NaN, and 0 otherwise
```

New mathematical functions found in C99 that are not found in C90 include:

```
double acosh(double x); // hyperbolic arccosine of x
double asinh(double x); // hyperbolic arcsine of x
double atanh(double x); // hyperbolic arctangent of x
double erf(double x); // returns the error function of x
double round(double x); // returns x rounded to the nearest integer
double tgamma(double x); // returns the gamma function of x
```

C99 supports the new mathematical functions for all real floating-point types.

Single precision versions of all existing `<math.h>` functions are also supported.

Related concepts

4.60 New library features of C99 on page 4-182.
4.75 Complex numbers in C99 on page 4-197.
4.76 Boolean type and <stdbool.h> in C99 on page 4-198.
4.77 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99 on page 4-199.
4.78 <fenv.h> floating-point environment access in C99 on page 4-200.
4.79 <stdio.h> snprintf family of functions in C99 on page 4-201.
4.80 <tgmath.h> type-generic math macros in C99 on page 4-202.
4.81 <wchar.h> wide character I/O functions in C99 on page 4-203.

Related information

Institute of Electrical and Electronics Engineers.

4.75 Complex numbers in C99

In C99 mode, the compiler supports complex and imaginary numbers. In GNU mode, the compiler supports complex numbers only.

For example:

```
#include <stdio.h>
#include <complex.h>
int main(void)
{
    complex float z = 64.0 + 64.0*I;
    printf("z = %f + %fI\n", creal(z), cimag(z));
    return 0;
}
```

The complex types are:

- **`float complex`**.
- **`double complex`**.
- **`long double complex`**.

Related concepts

4.59 New language features of C99 on page 4-180.
4.61 // comments in C99 and C90 on page 4-183.
4.62 Compound literals in C99 on page 4-184.
4.63 Designated initializers in C99 on page 4-185.
4.64 Hexadecimal floating-point numbers in C99 on page 4-186.
4.65 Flexible array members in C99 on page 4-187.
4.66 __func__ predefined identifier in C99 on page 4-188.
4.67 inline functions in C99 on page 4-189.
4.68 long long data type in C99 and C90 on page 4-190.
4.69 Macros with a variable number of arguments in C99 on page 4-191.
4.70 Mixed declarations and statements in C99 on page 4-192.
4.71 New block scopes for selection and iteration statements in C99 on page 4-193.
4.72 _Pragma preprocessing operator in C99 on page 4-194.
4.73 Restricted pointers in C99 on page 4-195.
4.60 New library features of C99 on page 4-182.
4.74 Additional <math.h> library functions in C99 on page 4-196.
4.76 Boolean type and <stdbool.h> in C99 on page 4-198.
4.77 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99 on page 4-199.
4.78 <fenv.h> floating-point environment access in C99 on page 4-200.
4.79 <stdio.h> snprintf family of functions in C99 on page 4-201.
4.80 <tgmath.h> type-generic math macros in C99 on page 4-202.
4.81 <wchar.h> wide character I/O functions in C99 on page 4-203.

4.76 Boolean type and <stdbool.h> in C99

C99 introduces the native type **`_Bool`**.

The associated standard header `<stdbool.h>` introduces the macros `bool`, `true` and `false` for Boolean tests. For example:

```
#include <stdbool.h>
bool foo(FILE *str)
{
    bool err = false;
    ...
    if (!fflush(str))
    {
        err = true;
    }
    ...
    return err;
}
```

Note

The C99 semantics for bool are intended to match those of C++.

Related concepts

4.60 New library features of C99 on page 4-182.
4.74 Additional <math.h> library functions in C99 on page 4-196.
4.75 Complex numbers in C99 on page 4-197.
4.77 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99 on page 4-199.
4.78 <fenv.h> floating-point environment access in C99 on page 4-200.
4.79 <stdio.h> snprintf family of functions in C99 on page 4-201.
4.80 <tgmath.h> type-generic math macros in C99 on page 4-202.
4.81 <wchar.h> wide character I/O functions in C99 on page 4-203.

4.77 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99

In C90, the **`long`** data type can serve both as the largest integral type, and as a 32-bit container. C99 removes this ambiguity through the new standard library header files `<inttypes.h>` and `<stdint.h>`.

The header file `<stdint.h>` introduces the new types:

- `intmax_t` and `uintmax_t`, that are maximum width signed and unsigned integer types.
- `intptr_t` and `unintptr_t`, that are integer types capable of holding signed and unsigned object pointers.

The header file `<inttypes.h>` provides library functions for manipulating values of type `intmax_t`, including:

```
intmax_t imaxabs(intmax_t x); // absolute value of x
imaxdiv_t imaxdiv(intmax_t x, intmax_t y) // returns the quotient and remainder
                                          // of x / y
```

These header files are also available in C90 and C++.

Related concepts

4.60 New library features of C99 on page 4-182.

4.74 Additional <math.h> library functions in C99 on page 4-196.

4.75 Complex numbers in C99 on page 4-197.

4.76 Boolean type and <stdbool.h> in C99 on page 4-198.

4.78 <fenv.h> floating-point environment access in C99 on page 4-200.

4.79 <stdio.h> snprintf family of functions in C99 on page 4-201.

4.80 <tgmath.h> type-generic math macros in C99 on page 4-202.

4.81 <wchar.h> wide character I/O functions in C99 on page 4-203.

4.78 <fenv.h> floating-point environment access in C99

The C99 standard header file `<fenv.h>` provides access to an IEEE 754-compliant floating-point environment for numerical programming.

The library introduces two types and numerous macros and functions for managing and controlling floating-point state.

The new types supported are:

- `fenv_t`, representing the entire floating-point environment.
- `fexcept_t`, representing the floating-point state.

New macros supported include:

- `FE_DIVBYZERO`, `FE_INEXACT`, `FE_INVALID`, `FE_OVERFLOW` and `FE_UNDERFLOW` for managing floating-point exceptions.
- `FE_DOWNWARD`, `FE_TONEAREST`, `FE_TOWARDZERO`, `FE_UPWARD` for managing rounding in the represented rounding direction.
- `FE_DFL_ENV`, representing the default floating-point environment.

New functions include:

```
int feclearexcept(int ex); // clear floating-point exceptions selected by ex
int feraiseexcept(int ex); // raise floating point exceptions selected by ex
int fetestexcept(int ex); // test floating point exceptions selected by ex
int fegetround(void); // return the current rounding mode
int fesetround(int mode); // set the current rounding mode given by mode
int fegetenv(fenv_t *penv); return the floating-point environment in penv
int fesetenv(const fenv_t *penv); // set the floating-point environment to penv
```

Related concepts

4.60 New library features of C99 on page 4-182.

4.74 Additional <math.h> library functions in C99 on page 4-196.

4.75 Complex numbers in C99 on page 4-197.

4.76 Boolean type and <stdbool.h> in C99 on page 4-198.

4.77 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99 on page 4-199.

4.79 <stdio.h> snprintf family of functions in C99 on page 4-201.

4.80 <tgmath.h> type-generic math macros in C99 on page 4-202.

4.81 <wchar.h> wide character I/O functions in C99 on page 4-203.

Related information

Institute of Electrical and Electronics Engineers.

4.79 <stdio.h> snprintf family of functions in C99

Using the `sprintf` family of functions found in the C90 standard header `<stdio.h>` can be dangerous.

In the statement:

```
sprintf(buffer, size, "Error %d: Cannot open file '%s'", errno, filename);
```

the variable `size` specifies the minimum number of characters to be inserted into `buffer`. Consequently, more characters can be output than might fit in the memory allocated to the string.

The `snprintf` functions found in the C99 version of `<stdio.h>` are safe versions of the `sprintf` functions that prevent buffer overrun. In the statement:

```
snprintf(buffer, size, "Error %d: Cannot open file '%s'", errno, filename);
```

the variable `size` specifies the maximum number of characters that can be inserted into `buffer`. The buffer can never be overrun, provided its size is always greater than the size specified by `size`.

Related concepts

4.60 New library features of C99 on page 4-182.
4.74 Additional <math.h> library functions in C99 on page 4-196.
4.75 Complex numbers in C99 on page 4-197.
4.76 Boolean type and <stdbool.h> in C99 on page 4-198.
4.77 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99 on page 4-199.
4.78 <fenv.h> floating-point environment access in C99 on page 4-200.
4.80 <tgmath.h> type-generic math macros in C99 on page 4-202.
4.81 <wchar.h> wide character I/O functions in C99 on page 4-203.

4.80 <tgmath.h> type-generic math macros in C99

The new standard header `<tgmath.h>` defines several families of mathematical functions that are type generic in the sense that they are overloaded on floating-point types.

For example, the trigonometric function `cos` works as if it has the overloaded declaration:

```
extern float cos(float x);
extern double cos(double x);
extern long double cos(long double x);
...
```

A statement such as:

```
p = cos(0.78539f); // p = cos(pi / 4)
```

calls the single-precision version of the `cos` function, as determined by the type of the literal `0.78539f`.

Note

Type-generic families of mathematical functions can be defined in C++ using the operator overloading mechanism. The semantics of type-generic families of functions defined using operator overloading in C++ are different from the semantics of the corresponding families of type-generic functions defined in `<tgmath.h>`.

Related concepts

4.60 New library features of C99 on page 4-182.
4.74 Additional <math.h> library functions in C99 on page 4-196.
4.75 Complex numbers in C99 on page 4-197.
4.76 Boolean type and <stdbool.h> in C99 on page 4-198.
4.77 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99 on page 4-199.
4.78 <fenv.h> floating-point environment access in C99 on page 4-200.
4.79 <stdio.h> snprintf family of functions in C99 on page 4-201.
4.81 <wchar.h> wide character I/O functions in C99 on page 4-203.

4.81 <wchar.h> wide character I/O functions in C99

Wide character I/O functions have been incorporated into C99. These enable you to read and write wide characters from a file in much the same way as normal characters.

The ARM C Library supports all of the C99 functions defined in `wchar.h`.

Related concepts

4.60 New library features of C99 on page 4-182.

4.74 Additional <math.h> library functions in C99 on page 4-196.

4.75 Complex numbers in C99 on page 4-197.

4.76 Boolean type and <stdbool.h> in C99 on page 4-198.

4.77 Extended integer types and functions in <inttypes.h> and <stdint.h> in C99 on page 4-199.

4.78 <fenv.h> floating-point environment access in C99 on page 4-200.

4.79 <stdio.h> snprintf family of functions in C99 on page 4-201.

4.80 <tgmath.h> type-generic math macros in C99 on page 4-202.

4.82 How to prevent uninitialized data from being initialized to zero

The ANSI C specification states that static data that is not explicitly initialized, is to be initialized to zero.

Therefore, by default, the compiler puts both zero-initialized and uninitialized data into the same ZI data section, which is populated with zeroes at runtime by the C library initialization code.

You can prevent uninitialized data from being initialized to zero by placing that data in a different section. This can be achieved using `#pragma arm section`, or with the GNU compiler extension `__attribute__((section("name")))`.

The following example shows how to retain uninitialized data using `#pragma arm section`:

```
#pragma arm section zidata = "non_initialized"
int i, j; // uninitialized data in non_initialized section (without the pragma,
             would be in .bss section by default)
#pragma arm section zidata // back to default (.bss section)
int k = 0, l = 0; // zero-initialized data in .bss section
```

The `non_initialized` section is placed into its own `UNINIT` execution region, as follows:

```
LOAD_1 0x0
{
  EXEC_1 +0
  {
    * (+RO)
    * (+RW)
    * (+ZI) ; ZI data gets initialized to zero
  }
  EXEC_2 +0 UNINIT
  {
    * (non_init) ; ZI data does not get initialized to zero
  }
}
```

Related references

9.77 #pragma arm section [section_type_list] on page 9-596.

9.67 __attribute__((section("name"))) variable attribute on page 9-586.

Related information

Execution region attributes.

Chapter 5
Compiler Diagnostic Messages

Describes the format of compiler diagnostic messages and how to control the output during compilation.

The compiler issues messages about potential portability problems and other hazards. It is possible to:
- Turn off specific messages. For example, warnings can be turned off if you are in the early stages of porting a program written in old-style C. In general, however, it is better to check the code than to turn off messages.
- Change the severity of specific messages.

It contains the following sections:

- *5.1 Severity of compiler diagnostic messages* on page 5-206.
- *5.2 Options that change the severity of compiler diagnostic messages* on page 5-207.
- *5.3 Controlling compiler diagnostic messages with pragmas* on page 5-209.
- *5.4 Prefix letters in compiler diagnostic messages* on page 5-211.
- *5.5 Compiler exit status codes and termination messages* on page 5-212.
- *5.6 Compiler data flow warnings* on page 5-213.

5.1 Severity of compiler diagnostic messages

Diagnostic messages have an associated *severity*.

The following table describes each of the different severities.

Table 5-1 Severity of diagnostic messages

Severity	Description
Internal fault	Internal faults indicate an internal problem with the compiler. Contact your supplier with feedback.
Error	Errors indicate problems that cause the compilation to stop. These errors include command line errors, internal errors, missing include files, and violations in the syntactic or semantic rules of the C or C++ language. If multiple source files are specified, no more source files are compiled.
Warning	Warnings indicate unusual conditions in your code that might indicate a problem. Compilation continues, and object code is generated unless any more problems with an Error severity are detected.
Remark	Remarks indicate common, but sometimes unconventional, use of C or C++. These diagnostics are not displayed by default. Compilation continues, and object code is generated unless any more problems with an Error severity are detected.

Related concepts

5.2 Options that change the severity of compiler diagnostic messages on page 5-207.

5.3 Controlling compiler diagnostic messages with pragmas on page 5-209.

5.4 Prefix letters in compiler diagnostic messages on page 5-211.

5.5 Compiler exit status codes and termination messages on page 5-212.

5.6 Compiler data flow warnings on page 5-213.

5.2 Options that change the severity of compiler diagnostic messages

You can change the diagnostic severity of all remarks and warnings, and a limited number of errors.

These options let you change severities:

`--diag_error=`*`tag`*`[, `*`tag`*`, ...]`
: Sets the diagnostic messages that have the specified tag, or tags, to Error severity.

`--diag_error=warning`
: Upgrades all warning messages to Error severity.

`--diag_remark=`*`tag`*`[, `*`tag`*`, ...]`
: Sets the diagnostic messages that have the specified tag, or tags, to Remark severity.

`--diag_warning=`*`tag`*`[, `*`tag`*`, ...]`
: Sets the diagnostic messages that have the specified tag, or tags, to Warning severity.

`--diag_warning=error`
: Sets all downgradable error messages to Warning severity.

The format *`tag`*`[, `*`tag`*`, ...]` indicates a comma-separated list of the error messages that you want to change. For example, you might want to change a warning message with the number `1293` to Remark severity, because remarks are not displayed by default.

Note

`tag` is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.

To do this, use the following command:

```
armcc --diag_remark=1293 ...
```

Only errors with a suffix of `-D` following the error number can be downgraded by changing them into warnings or remarks.

Note

These options also have pragma equivalents.

The following diagnostic messages can be changed:

- Messages with the number format `#`*`nnnn`*`-D`.
- Warning messages with the number format `C`*`nnnn`*`W`.

It is also possible to apply changes to optimization messages as a group. For example, `--diag_warning=optimizations`. By default, optimization messages are remarks.

Related concepts

5.3 Controlling compiler diagnostic messages with pragmas on page 5-209.
5.4 Prefix letters in compiler diagnostic messages on page 5-211.
5.5 Compiler exit status codes and termination messages on page 5-212.
5.6 Compiler data flow warnings on page 5-213.

Related references

5.1 Severity of compiler diagnostic messages on page 5-206.
9.78 #pragma diag_default tag[,tag,...] on page 9-598.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
9.82 #pragma diag_warning tag[, tag, ...] on page 9-602.

9.96 #pragma pop on page 9-617.
9.97 #pragma push on page 9-618.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.49 --diag_warning=optimizations on page 7-323.

5.3 Controlling compiler diagnostic messages with pragmas

Pragmas let you suppress, enable, or change the severity of specific diagnostic messages from within your code.

For example, you can suppress a particular diagnostic message when compiling one specific function.

Note

You can alternatively use command-line options to suppress or change the severity of messages, but the change applies for the entire compilation.

Examples

The following example shows three identical functions, `foo1()`, `foo2()`, and `foo3()`, all of which would normally provoke diagnostic message `#177-D: variable "x" was declared but never referenced`.

For `foo1()`, the current pragma state is pushed to the stack and `#pragma diag_suppress` suppresses the message. The message is re-enabled by `#pragma pop` before compiling `foo2()`. In `foo3()`, the message is not suppressed because the `#pragma push` and `#pragma pop` do not enclose the full scope responsible for the generation of the message:

```
#pragma push
#pragma diag_suppress 177
void foo1( void )
{
    /* Here we do not expect a diagnostic, because we suppressed it. */
    int x;
}
#pragma pop

void foo2( void )
{
    /* Here we do, because the suppression was inside push/pop. */
    int x;
}

void foo3( void )
{
    #pragma push
    #pragma diag_suppress 177
    /* Here, the suppression fails because the push/pop must enclose the whole function. */
    int x;
    #pragma pop
}
```

Diagnostic messages use the pragma state in place at the time they are generated. If you use pragmas to control a message in your code, you must be aware of when that message is generated. For example, the following code is intended to suppress the diagnostic message `#177-D: function "dummy" was declared but never referenced`:

```
#include <stdio.h>
#pragma push
#pragma diag_suppress 177
static int dummy(void)
{
    printf("This function is never called.");
    return 1;
}
#pragma pop
main(void){
     printf("Hello world!\n");
}
```

However, message 177 is only generated after all functions have been processed. Therefore, the message is generated after `pragma pop` restores the pragma state, and message 177 is not suppressed.

Removing `pragma push` and `pragma pop` would correctly suppress message 177, but would suppress messages for all unreferenced functions rather than for only the `dummy()` function.

Related concepts

5.2 Options that change the severity of compiler diagnostic messages on page 5-207.
5.4 Prefix letters in compiler diagnostic messages on page 5-211.
5.5 Compiler exit status codes and termination messages on page 5-212.
5.6 Compiler data flow warnings on page 5-213.

Related references

5.1 Severity of compiler diagnostic messages on page 5-206.
9.78 #pragma diag_default tag[,tag,...] on page 9-598.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
9.82 #pragma diag_warning tag[, tag, ...] on page 9-602.
9.96 #pragma pop on page 9-617.
9.97 #pragma push on page 9-618.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.49 --diag_warning=optimizations on page 7-323.

5.4 Prefix letters in compiler diagnostic messages

The compilation tools automatically insert an identification letter to diagnostic messages.

The following table shows the prefix letters used by the compilation tools. Using these prefix letters enables the tools to use overlapping message ranges.

Table 5-2 Identifying diagnostic messages

Prefix letter	Tool
`C`	`armcc`
`A`	`armasm`
`L`	`armlink` or `armar`
`Q`	`fromelf`

The following rules apply:
- All of the compilation tools act on a message number without a prefix.
- A message number with a prefix is only acted on by the tool with the matching prefix.
- A tool does not act on a message with a non-matching prefix.

Therefore, the compiler prefix `C` can be used with `--diag_error`, `--diag_remark`, and `--diag_warning`, or when suppressing messages, for example:

```
armcc --diag_suppress=C1287,C4017 ...
```

Use the prefix letters to control options that are passed from the compiler to other tools, for example, include the prefix letter `L` to specify linker message numbers.

Related concepts

5.2 Options that change the severity of compiler diagnostic messages on page 5-207.
5.3 Controlling compiler diagnostic messages with pragmas on page 5-209.
5.5 Compiler exit status codes and termination messages on page 5-212.
5.6 Compiler data flow warnings on page 5-213.

Related references

5.1 Severity of compiler diagnostic messages on page 5-206.

5.5 Compiler exit status codes and termination messages

If the compiler detects any warnings or errors during compilation, it writes the messages to `stderr`.

At the end of the messages, a summary message is displayed that gives the total number of each type of message of the form:

```
filename: n warnings, n errors
```

where *n* indicates the number of warnings or errors detected.

——— **Note** ———

Remarks are not displayed by default. To display remarks, use the `--remarks` compiler option. No summary message is displayed if only remark messages are generated.

———————

The signals **SIGINT** (caused by a user interrupt, like ^C) and **SIGTERM** (caused by a UNIX `kill` command) are trapped by the compiler and cause abnormal termination.

On completion, the compiler returns a value greater than zero if an error is detected. If no error is detected, a value of zero is returned.

Related concepts

5.2 Options that change the severity of compiler diagnostic messages on page 5-207.
5.3 Controlling compiler diagnostic messages with pragmas on page 5-209.
5.4 Prefix letters in compiler diagnostic messages on page 5-211.
5.6 Compiler data flow warnings on page 5-213.

Related references

5.1 Severity of compiler diagnostic messages on page 5-206.

5.6 Compiler data flow warnings

The compiler performs data flow analysis as part of its optimization process. This information can help identify potential problems in your code, for example, issuing warnings about the use of uninitialized variables.

The data flow analysis can only warn about local variables that are held in processor registers, not global variables held in memory or variables or structures that are placed on the stack.

Be aware that:

- In ARM Compiler 5.04 and later, data flow warnings are suppressed by default. To output them, use the `--diag_warning=4017` option. In *RealView Compiler Tools* (RVCT) v2.0 and earlier, data flow warnings are issued only if you specify the `-fa` option.
- Data flow analysis is disabled at optimization level `-O0`, even if you specify `--diag_warning=4017`.

For example, the following code produces the warning `C4017W: i may be used before being set`, if you have enabled it, when compiling at `-O1` and above:

```
int f(void)
{
    int i;
    return i++;
}
```

The results of the analysis vary with the level of optimization used. This means that higher optimization levels might produce a number of warnings that do not appear at lower levels.

The data flow analysis cannot reliably identify faulty code and any `C4017W` warnings issued by the compiler are intended only as an indication of possible problems. For a full analysis of your code, use an appropriate third-party analysis tool, for example Lint.

Related concepts

5.2 Options that change the severity of compiler diagnostic messages on page 5-207.

5.3 Controlling compiler diagnostic messages with pragmas on page 5-209.

5.4 Prefix letters in compiler diagnostic messages on page 5-211.

5.5 Compiler exit status codes and termination messages on page 5-212.

Related references

5.1 Severity of compiler diagnostic messages on page 5-206.

Chapter 6
Using the Inline and Embedded Assemblers of the ARM Compiler

Describes the optimizing inline assembler and non-optimizing embedded assembler of the ARM compiler, `armcc`.

Note

Using intrinsics is generally preferable to using inline or embedded assembly language.

It contains the following sections:

- *6.1 Compiler support for inline assembly language* on page 6-216.
- *6.2 Inline assembler support in the compiler* on page 6-217.
- *6.3 Restrictions on inline assembler support in the compiler* on page 6-218.
- *6.4 Inline assembly language syntax with the __asm keyword in C and C++* on page 6-219.
- *6.5 Inline assembly language syntax with the asm keyword in C++* on page 6-220.
- *6.6 Inline assembler rules for compiler keywords __asm and asm* on page 6-221.
- *6.7 Restrictions on inline assembly operations in C and C++ code* on page 6-222.
- *6.8 Inline assembler register restrictions in C and C++ code* on page 6-223.
- *6.9 Inline assembler processor mode restrictions in C and C++ code* on page 6-224.
- *6.10 Inline assembler Thumb instruction set restrictions in C and C++ code* on page 6-225.
- *6.11 Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code* on page 6-226.
- *6.12 Inline assembler instruction restrictions in C and C++ code* on page 6-227.
- *6.13 Miscellaneous inline assembler restrictions in C and C++ code* on page 6-228.
- *6.14 Inline assembler and register access in C and C++ code* on page 6-229.
- *6.15 Inline assembler and the # constant expression specifier in C and C++ code* on page 6-231.
- *6.16 Inline assembler and instruction expansion in C and C++ code* on page 6-232.

- *6.17 Expansion of inline assembler instructions that use constants* on page 6-233.
- *6.18 Expansion of inline assembler load and store instructions* on page 6-234.
- *6.19 Inline assembler effect on processor condition flags in C and C++ code* on page 6-235.
- *6.20 Inline assembler expression operands in C and C++ code* on page 6-236.
- *6.21 Inline assembler register list operands in C and C++ code* on page 6-237.
- *6.22 Inline assembler intermediate operands in C and C++ code* on page 6-238.
- *6.23 Inline assembler function calls and branches in C and C++ code* on page 6-239.
- *6.24 Inline assembler branches and labels in C and C++ code* on page 6-241.
- *6.25 Inline assembler and virtual registers* on page 6-242.
- *6.26 Embedded assembler support in the compiler* on page 6-243.
- *6.27 Embedded assembler syntax in C and C++* on page 6-244.
- *6.28 Effect of compiler ARM and Thumb states on embedded assembler* on page 6-245.
- *6.29 Restrictions on embedded assembly language functions in C and C++ code* on page 6-246.
- *6.30 Compiler generation of embedded assembly language functions* on page 6-247.
- *6.31 Access to C and C++ compile-time constant expressions from embedded assembler* on page 6-249.
- *6.32 Differences between expressions in embedded assembler and C or C++* on page 6-250.
- *6.33 Manual overload resolution in embedded assembler* on page 6-251.
- *6.34 __offsetof_base keyword for related base classes in embedded assembler* on page 6-252.
- *6.35 Compiler-supported keywords for calling class member functions in embedded assembler* on page 6-253.
- *6.36 __mcall_is_virtual(D, f)* on page 6-254.
- *6.37 __mcall_is_in_vbase(D, f)* on page 6-255.
- *6.38 __mcall_offsetof_vbase(D, f)* on page 6-256.
- *6.39 __mcall_this_offset(D, f)* on page 6-257.
- *6.40 __vcall_offsetof_vfunc(D, f)* on page 6-258.
- *6.41 Calling nonstatic member functions in embedded assembler* on page 6-259.
- *6.42 Calling a nonvirtual member function* on page 6-260.
- *6.43 Calling a virtual member function* on page 6-261.
- *6.44 Accessing sp (r13), lr (r14), and pc (r15)* on page 6-262.
- *6.45 Differences in compiler support for inline and embedded assembly code* on page 6-263.

6.1 Compiler support for inline assembly language

The compiler provides an inline assembler that enables you to write optimized assembly language routines, and to access features of the target processor not available from C or C++.

Related concepts

6.2 Inline assembler support in the compiler on page 6-217.
6.3 Restrictions on inline assembler support in the compiler on page 6-218.
6.4 Inline assembly language syntax with the __asm keyword in C and C++ on page 6-219.
6.5 Inline assembly language syntax with the asm keyword in C++ on page 6-220.
6.6 Inline assembler rules for compiler keywords __asm and asm on page 6-221.
6.7 Restrictions on inline assembly operations in C and C++ code on page 6-222.
6.14 Inline assembler and register access in C and C++ code on page 6-229.
6.15 Inline assembler and the # constant expression specifier in C and C++ code on page 6-231.
6.19 Inline assembler effect on processor condition flags in C and C++ code on page 6-235.
6.20 Inline assembler expression operands in C and C++ code on page 6-236.
6.21 Inline assembler register list operands in C and C++ code on page 6-237.
6.22 Inline assembler intermediate operands in C and C++ code on page 6-238.
6.45 Differences in compiler support for inline and embedded assembly code on page 6-263.
6.23 Inline assembler function calls and branches in C and C++ code on page 6-239.
6.24 Inline assembler branches and labels in C and C++ code on page 6-241.
6.16 Inline assembler and instruction expansion in C and C++ code on page 6-232.

Related references

9.156 Named register variables on page 9-685.

Related information

armasm User Guide.

6.2 Inline assembler support in the compiler

The inline assembler supports ARM assembly language for all architectures, and Thumb assembly language in ARMv6T2, ARMv6M, and ARMv7.

For ARMv7, the inline assembler supports:
- Most ARM instructions.
- Most Thumb instructions.

For ARMv6T2, the inline assembler supports most Thumb instructions.

For ARMv6, the inline assembler supports most ARM instructions, including the complete set of ARMv6 *Single Instruction Multiple Data* (SIMD) instructions.

For ARMv5, the inline assembler supports most ARM instructions, including generic coprocessor instructions.

For ARMv4, the inline assembler supports most ARM instructions, including generic coprocessor instructions.

VFPv2 instructions are supported in the inline assembler.

6.3 Restrictions on inline assembler support in the compiler

The inline assembler in the compiler does not support a number of instructions.

Specifically, the inline assembler does not support:

- Thumb assembly language in processors without Thumb-2 technology.
- VFP instructions that were added in VFPv3 or higher.
- The ARMv6 `SETEND` instruction and some of the system extensions.
- ARMv5 `BX`, `BLX`, and `BXJ` instructions.

6.4 Inline assembly language syntax with the __asm keyword in C and C++

The inline assembler is invoked with the assembler specifier, `__asm`, and is followed by a list of assembler instructions inside braces or parentheses.

You can specify inline assembly code using the following formats:

- On a single line, for example:

  ```
  __asm("instruction[;instruction]");
  __asm{instruction[;instruction]}
  ```

  You cannot include comments.
- Using multiple adjacent strings, for example:

  ```
  __asm("ADD x, x, #1\n"
        "MOV y, x\n");
  ```

  This enables you to use macros to generate inline assembly, for example:

  ```
  #define ADDLSL(x, y, shift) __asm ("ADD " #x ", " #y ", LSL " #shift)
  ```
- On multiple lines, for example:

  ```
  __asm
  {
      ...
      instruction
      ...
  }
  ```

  You can use C or C++ comments anywhere in an inline assembly language block.

You can use an `__asm` statement wherever a statement is expected.

6.5 Inline assembly language syntax with the asm keyword in C++

When compiling C++, the compiler supports the **`asm`** syntax proposed in the ISO C++ Standard.

You can specify inline assembly code using the following formats:

- On a single line, for example:

  ```
  asm("instruction[;instruction]");
  asm{instruction[;instruction]}
  ```

  You cannot include comments.
- Using multiple adjacent strings, for example:

  ```
  asm("ADD x, x, #1\n"
        "MOV y, x\n");
  ```

  This enables you to use macros to generate inline assembly, for example:

  ```
  #define ADDLSL(x, y, shift) asm ("ADD " #x ", " #y ", LSL " #shift)
  ```
- On multiple lines, for example:

  ```
  asm
  {
      ...
      instruction
      ...
  }
  ```

  You can use C or C++ comments anywhere in an inline assembly language block.

You can use an `asm` statement wherever a statement is expected.

6.6 Inline assembler rules for compiler keywords __asm and asm

There are a number of rule that apply to the **`__asm`** and **`asm`** keywords.

These rules are as follows:

- Multiple instructions on the same line must be separated with a semicolon (`;`).
- If an instruction requires more than one line, line continuation must be specified with the backslash character (\).
- For the multiple line format, C and C++ comments are permitted anywhere in the inline assembly language block. However, comments cannot be embedded in a line that contains multiple instructions.
- The comma (`,`) is used as a separator in assembly language, so C expressions with the comma operator must be enclosed in parentheses to distinguish them:

```
__asm
{
    ADD x, y, (f(), z)
}
```

- Labels must be followed by a colon, `:`, like C and C++ labels.
- An **`asm`** statement must be inside a C++ function. An **`asm`** statement can be used anywhere a C++ statement is expected.
- Register names in inline assembly code are treated as C or C++ variables. They do not necessarily relate to the physical register of the same name. If the register is not declared as a C or C++ variable, the compiler generates a warning.
- Registers must not be saved and restored in inline assembly code. The compiler does this for you. Also, the inline assembler does not provide direct access to the physical registers. However, indirect access is provided through variables that act as virtual registers.

  If registers other than `ASPR`, `CPSR`, and `SPSR` are read without being written to, an error message is issued. For example:

```
int f(int x)
{
    __asm
    {
        STMFD sp!, {r0}    // save r0 - illegal: read before write
        ADD r0, x, 1
        EOR x, r0, x
        LDMFD sp!, {r0}    // restore r0 - not needed.
    }
    return x;
}
```

  The function must be written as:

```
int f(int x)
{
    int r0;
    __asm
    {
        ADD r0, x, 1
        EOR x, r0, x
    }
    return x;
}
```

6.7 Restrictions on inline assembly operations in C and C++ code

There are a number of restrictions on the operations that can be performed in inline assembly code.

These restrictions provide a measure of safety, and ensure that the assumptions in compiled C and C++ code are not violated in the assembled assembly code.

Related concepts

6.8 Inline assembler register restrictions in C and C++ code on page 6-223.

6.9 Inline assembler processor mode restrictions in C and C++ code on page 6-224.

6.10 Inline assembler Thumb instruction set restrictions in C and C++ code on page 6-225.

6.11 Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code on page 6-226.

6.12 Inline assembler instruction restrictions in C and C++ code on page 6-227.

6.13 Miscellaneous inline assembler restrictions in C and C++ code on page 6-228.

6.8 Inline assembler register restrictions in C and C++ code

Registers such as `r0-r3`, `sp`, `lr`, and the NZCV flags in the `CPSR` must be used with caution.

If C or C++ expressions are used, these might be used as temporary registers and NZCV flags might be corrupted by the compiler when evaluating the expression.

The `pc`, `lr`, and `sp` registers cannot be explicitly read or modified using inline assembly code because there is no direct access to any physical registers. However, you can use the intrinsics `__current_pc`, `__current_sp`, and `__return_address` to read these registers.

Related concepts

6.7 Restrictions on inline assembly operations in C and C++ code on page 6-222.
6.9 Inline assembler processor mode restrictions in C and C++ code on page 6-224.
6.10 Inline assembler Thumb instruction set restrictions in C and C++ code on page 6-225.
6.11 Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code on page 6-226.
6.12 Inline assembler instruction restrictions in C and C++ code on page 6-227.
6.13 Miscellaneous inline assembler restrictions in C and C++ code on page 6-228.
6.14 Inline assembler and register access in C and C++ code on page 6-229.

Related references

9.108 __current_pc intrinsic on page 9-630.
9.109 __current_sp intrinsic on page 9-631.
9.134 __return_address intrinsic on page 9-659.

6.9 Inline assembler processor mode restrictions in C and C++ code

ARM strongly recommends that you do not change processor modes or modify coprocessor states in inline assembly code.

——— **Caution** ———

The compiler does not recognize such changes.

———————————

Instead of attempting to change processor modes or coprocessor states from within inline assembly code, see if there are any intrinsics available that provide what you require. If no such intrinsics are available, use embedded assembly code if absolutely necessary.

Related concepts

6.7 Restrictions on inline assembly operations in C and C++ code on page 6-222.

6.8 Inline assembler register restrictions in C and C++ code on page 6-223.

6.10 Inline assembler Thumb instruction set restrictions in C and C++ code on page 6-225.

6.11 Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code on page 6-226.

6.12 Inline assembler instruction restrictions in C and C++ code on page 6-227.

6.13 Miscellaneous inline assembler restrictions in C and C++ code on page 6-228.

3.1 Compiler intrinsics on page 3-64.

6.26 Embedded assembler support in the compiler on page 6-243.

Related information

Processor modes, and privileged and unprivileged software execution.

6.10 Inline assembler Thumb instruction set restrictions in C and C++ code

The inline assembler supports Thumb state in ARM architectures v6T2, v6M, and v7. There are a number of Thumb-specific restrictions.

These restrictions are as follows:
1. `TBB`, `TBH`, `CBZ`, and `CBNZ` instructions are not supported.
2. In some cases, the compiler can replace `IT` blocks with branched code.
3. The instruction width specifier `.N` denotes a preference, but not a requirement, to the compiler. This is because, in rare cases, optimizations and register allocation can make it inefficient to generate a 16-bit encoding.

For ARMv6 and lower architectures, the inline assembler does not assemble any Thumb instructions. Instead, on finding inline assembly while in Thumb state, the compiler switches to ARM state automatically. Code that relies on this switch is currently supported, but this practise is deprecated. For ARMv6T2 and higher, the automatic switch from Thumb to ARM state is made if the code is valid ARM assembly but not Thumb.

ARM state can be set deliberately. Inline assembly language can be included in a source file that contains code to be compiled for Thumb in ARMv6 and lower, by enclosing the functions containing inline assembly code between `#pragma arm` and `#pragma thumb` statements. For example:

```
...         // Thumb code
#pragma arm // ARM code. Switch code generation to the ARM instruction set so
            // that the inline assembler is available for Thumb in ARMv6 and lower.
int add(int i, int j)
{
    int res;
    __asm
    {
        ADD   res, i, j   // add here
    }
    return res;
}
#pragma thumb   // Thumb code. Switch back to the Thumb instruction set.
                // The inline assembler is no longer available for Thumb in ARMv6 and
                // lower.
```

The code must also be compiled using the `--apcs /interwork` compiler command-line option.

Related concepts

6.7 Restrictions on inline assembly operations in C and C++ code on page 6-222.
6.8 Inline assembler register restrictions in C and C++ code on page 6-223.
6.9 Inline assembler processor mode restrictions in C and C++ code on page 6-224.
6.11 Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code on page 6-226.
6.12 Inline assembler instruction restrictions in C and C++ code on page 6-227.
6.13 Miscellaneous inline assembler restrictions in C and C++ code on page 6-228.

Related references

7.6 --apcs=qualifier...qualifier on page 7-273.
9.74 Pragmas on page 9-593.

Related information

Instruction width specifiers.
IT.
TBB and TBH.
CBZ and CBNZ.

6.11 Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code

The inline assembler provides direct support for VFPv2 instructions.

For example:

```
float foo(float f, float g)
{
  float h;
  __asm
  {
      VADD h, f, 0.5*g; // h = f + 0.5*g
  }
  return h;
}
```

If you change the FPSCR register using inline assembly code, it produces runtime effects on the inline VFP code and on subsequent compiler-generated VFP code.

Note

- Do not use inline assembly code to change VFP vector mode. Inline assembly code must not be used for this purpose, and VFP vector mode is deprecated.
- ARM strongly discourages the use of inline assembly coprocessor instructions to interact with VFP in any way.

Related concepts

6.7 Restrictions on inline assembly operations in C and C++ code on page 6-222.
6.8 Inline assembler register restrictions in C and C++ code on page 6-223.
6.9 Inline assembler processor mode restrictions in C and C++ code on page 6-224.
6.10 Inline assembler Thumb instruction set restrictions in C and C++ code on page 6-225.
6.12 Inline assembler instruction restrictions in C and C++ code on page 6-227.
6.13 Miscellaneous inline assembler restrictions in C and C++ code on page 6-228.
4.41 Compiler support for floating-point arithmetic on page 4-154.

Related information

VMOV (between one ARM register and single precision VFP).

6.12 Inline assembler instruction restrictions in C and C++ code

There are a number of instructions that the inline assembler does not support.

Specifically, the following instructions are not supported:

- `BKPT`, `BX`, `BXJ`, and `BLX` instructions.

  Note

  You can insert a `BKPT` instruction in C and C++ code by using the `__breakpoint()` intrinsic.

- `LDR R`*`n`*`, =`*`expression`* pseudo-instruction. Use `MOV R`*`n`*`, `*`expression`* instead. (This can generate a load from a literal pool.)
- `LDRT`, `LDRBT`, `STRT`, and `STRBT` instructions.
- `MUL`, `MLA`, `UMULL`, `UMLAL`, `SMULL`, and `SMLAL` flag setting instructions.
- `MOV` or `MVN` flag-setting instructions where the second operand is a constant.
- The special `LDM` instructions used in system or supervisor mode to load the user-mode banked registers, written with a ^ after the register list, such as:

  ```
  LDMIA sp!, {r0-r12, lr, pc}^
  ```

- `ADR` and `ADRL` pseudo-instructions.

  Note

  You can use `MOV R`*`n`*`, &`*`expression`*`;` instead of the `ADR` and `ADRL` pseudo-instructions.

- ARM recommends not using the `LDREX` and `STREX` instructions. This is because the compiler might generate loads and stores between `LDREX` and `STREX`, potentially clearing the exclusive monitor set by `LDREX`. This recommendation also applies to the byte, halfword, and doubleword variants `LDREXB`, `STREXB`, `LDREXH`, `STREXH`, `LDREXD`, and `STREXD`.

Related concepts

6.7 Restrictions on inline assembly operations in C and C++ code on page 6-222.
6.8 Inline assembler register restrictions in C and C++ code on page 6-223.
6.9 Inline assembler processor mode restrictions in C and C++ code on page 6-224.
6.10 Inline assembler Thumb instruction set restrictions in C and C++ code on page 6-225.
6.11 Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code on page 6-226.
6.13 Miscellaneous inline assembler restrictions in C and C++ code on page 6-228.

Related references

9.104 __breakpoint intrinsic on page 9-626.

6.13 Miscellaneous inline assembler restrictions in C and C++ code

Compared with `armasm` or embedded assembly language, the inline assembler has a number of restrictions.

Specifically, these restrictions are as follows:

- The inline assembler is a high-level assembler, and the code it generates might not always be exactly what you write. Do not use it to generate more efficient code than the compiler generates. Use the embedded assembler or the ARM assembler `armasm` for this purpose.
- Some low-level features that are available in the ARM assembler `armasm`, such as writing to PC, are not supported.
- Label expressions are not supported.
- You cannot get the address of the current instruction using dot notation (.) or `{PC}`.
- You cannot use the `&` operator to denote hexadecimal constants. Use the `0x` prefix instead. For example:

```
__asm { AND x, y, 0xF00 }
```

- The notation to specify the actual rotation of an 8-bit constant is not available in inline assembly language. This means that where an 8-bit shifted constant is used, the C flag must be regarded as corrupted if the NZCV flags are updated.
- You must not modify the stack pointer. This is not necessary because the compiler automatically stacks and restores any working registers as required. The compiler does not permit you to explicitly stack and restore work registers.

Related concepts

6.7 Restrictions on inline assembly operations in C and C++ code on page 6-222.
6.8 Inline assembler register restrictions in C and C++ code on page 6-223.
6.9 Inline assembler processor mode restrictions in C and C++ code on page 6-224.
6.10 Inline assembler Thumb instruction set restrictions in C and C++ code on page 6-225.
6.11 Inline assembler Vector Floating-Point (VFP) restrictions in C and C++ code on page 6-226.
6.12 Inline assembler instruction restrictions in C and C++ code on page 6-227.

6.14 Inline assembler and register access in C and C++ code

The inline assembler provides no direct access to the physical registers of an ARM processor. If an ARM register name is used as an operand in an inline assembler instruction it becomes a reference to a variable of the same name, and not the physical ARM register.

The variable can be thought of as a virtual register.

The compiler declares variables for physical registers as appropriate during optimization and code generation. However, the physical register used in the assembled code might be different to that specified in the instruction, or it might be stored on the stack. You can explicitly declare variables representing physical registers as normal C or C++ variables. The compiler implicitly declares registers `R0` to `R12` and `r0` to `r12` as **`auto signed int`** local variables, regardless of whether or not they are used. If you want to declare them to be of a different data type, you can do so. For example, in the following code, the compiler does not implicitly declare `r1` and `r2` as **`auto signed int`** because they are explicitly declared as **`char`** and **`float`** types respectively:

```
void bar(float *);
int add(int x)
{
  int a = 0;
  char r1 = 0;
  float r2 = 0.0;
  bar(&r2);
  __asm
  {
    ADD r1, a, #100
  }
  ...
  return r1;
}
```

The compiler does not implicitly declare variables for any other registers, so you must explicitly declare variables for registers other than `R0` to `R12` and `r0` to `r12` in your C or C++ code. No variables are declared for the `sp` (`r13`), `lr` (`r14`), and `pc` (`r15`) registers, and they cannot be read or directly modified in inline assembly code.

There is no virtual *Processor Status Register* (PSR). Any references to the `PSR` are always to the physical `PSR`.

The size of the variables is the same as the physical registers.

The compiler-declared variables have function local scope, that is, within a single function, multiple **`asm`** statements or declarations that reference the same variable name access the same virtual register.

Existing inline assembly code that conforms to previously documented guidelines continues to perform the same function as in previous versions of the compiler, although the actual registers used in each instruction might be different.

The initial value in each variable representing a physical register is UNKNOWN. You must write to these variables before reading them. The compiler generates an error if you attempt to read such a variable before writing to it, for example, if you attempt to read the variable associated with the physical register `r1`.

Any variables that you use in inline assembly code to refer to registers must be explicitly declared in your C or C++ code, unless they are implicitly declared by the compiler. However, it is better to *explicitly* declare them in your C or C++ code. You do not have to declare them to be of the same data type as the implicit declarations. For example, although the compiler implicitly declares register `R0` to be of type **`signed int`**, you can explicitly declare `R0` as an unsigned integer variable if required.

It is also better to use C or C++ variables as instruction operands. The compiler generates a warning the first time a variable or physical register name is used, regardless of whether it is implicitly or explicitly declared, and only once for each translation unit. For example, if you use register `r3` without declaring it, a warning is displayed. You can suppress the warning with `--diag_suppress`.

Related concepts

6.18 Expansion of inline assembler load and store instructions on page 6-234.

6.8 Inline assembler register restrictions in C and C++ code on page 6-223.

Related references

7.46 --diag_suppress=tag[,tag,...] on page 7-320.

6.15 Inline assembler and the # constant expression specifier in C and C++ code

The constant expression specifier # is optional. If it is used, the expression following it must be a constant.

6.16 Inline assembler and instruction expansion in C and C++ code

An ARM instruction in inline assembly code might be expanded into several instructions in the compiled object.

The expansion depends on the instruction, the number of operands specified in the instruction, and the type and value of each operand.

Related concepts

6.17 Expansion of inline assembler instructions that use constants on page 6-233.

6.18 Expansion of inline assembler load and store instructions on page 6-234.

6.1 Compiler support for inline assembly language on page 6-216.

6.17 Expansion of inline assembler instructions that use constants

A constant operand specified in an instruction is not limited to the values permitted by the instruction. Instead, the compiler might translate the instruction into a sequence of instructions with the same effect.

For example:

```
ADD r0,r0,#1023
```

might be translated into:

```
ADD r0,r0,#1024
SUB r0,r0,#1
```

Another example of expansion possibility is:

```
MOV rn,0x12345678
```

With the exception of coprocessor instructions, all ARM instructions with a constant operand support instruction expansion. In addition, the `MUL` instruction can be expanded into a sequence of adds and shifts when the third operand is a constant.

The effect of updating the `CPSR` by an expanded instruction is:

- Arithmetic instructions set the NZCV flags correctly.
- Logical instructions:
  - Set the NZ flags correctly.
  - Do not change the V flag.
  - Corrupt the C flag.

Related concepts

6.16 Inline assembler and instruction expansion in C and C++ code on page 6-232.

6.18 Expansion of inline assembler load and store instructions

The `LDM`, `STM`, `LDRD`, and `STRD` instructions might be replaced by equivalent ARM instructions.

In this case the compiler outputs a warning message informing you that it might expand instructions. The warning can be suppressed with `--diag_suppress`.

Inline assembly code must be written in such a way that it does not depend on the number of expected instructions or on the expected execution time for each specified instruction.

Instructions that normally place constraints on pairs of operand registers, such as `LDRD` and `STRD`, are replaced by a sequence of instructions with equivalent functionality and without the constraints. However, these might be recombined into `LDRD` and `STRD` instructions.

All `LDM` and `STM` instructions are expanded into a sequence of `LDR` and `STR` instructions with equivalent effect. However, the compiler might subsequently recombine the separate instructions into an `LDM` or `STM` during optimization.

Related concepts

6.14 Inline assembler and register access in C and C++ code on page 6-229.

6.16 Inline assembler and instruction expansion in C and C++ code on page 6-232.

Related references

7.46 --diag_suppress=tag[,tag,...] on page 7-320.

6.19 Inline assembler effect on processor condition flags in C and C++ code

An inline assembly language instruction might explicitly or implicitly attempt to update the processor condition flags.

Inline assembly language instructions that involve only virtual register operands or simple expression operands have predictable behavior. The condition flags are set by the instruction if either an implicit or an explicit update is specified. The condition flags are unchanged if no update is specified.

If any of the instruction operands are not simple operands, then the condition flags might be corrupted unless the instruction updates them.

In general, the compiler cannot easily diagnose potential corruption of the condition flags. However, for operands that require the construction and subsequent destruction of C++ temporaries the compiler gives a warning if the instruction attempts to update the condition flags. This is because the destruction might corrupt the condition flags.

Related concepts

6.20 Inline assembler expression operands in C and C++ code on page 6-236.

6.21 Inline assembler register list operands in C and C++ code on page 6-237.

6.20 Inline assembler expression operands in C and C++ code

Function arguments, C or C++ variables, and other C or C++ expressions can be specified as register operands in an inline assembly language instruction.

The type of an expression used in place of an ARM integer register must be either an integral type (that is, **`char`**, **`short`**, **`int`** or **`long`**), excluding **`long long`**, or a pointer type. No sign extension is performed on **`char`** or **`short`** types. You must perform sign extension explicitly for these types. The compiler might add code to evaluate these expressions and allocate them to registers.

When an operand is used as a destination, the expression must be a modifiable lvalue if used as an operand where the register is modified. For example, a destination register or a base register with a base-register update.

For an instruction containing more than one expression operand, the order that expression operands are evaluated is unspecified.

An expression operand of a conditional instruction is only evaluated if the conditions for the instruction are met.

A C or C++ expression that is used as an inline assembly code operand might result in the instruction being expanded into several instructions. This happens if the value of the expression does not meet the constraints set out for the instruction operands in the *ARM Architecture Reference Manual*.

If an expression used as an operand creates a temporary that requires destruction, then the destruction occurs after the inline assembly instruction is executed. This is analogous to the C++ rules for destruction of temporaries.

A simple expression operand is one of the following:

- A variable value.
- The address of a variable.
- The dereferencing of a pointer variable.
- A compile-time constant.

Any expression containing one of the following is not a simple expression operand:

- An implicit function call, such as for division, or explicit function call.
- The construction of a C++ temporary.
- An arithmetic or logical operation.

Related concepts

6.21 Inline assembler register list operands in C and C++ code on page 6-237.

6.22 Inline assembler intermediate operands in C and C++ code on page 6-238.

Related information

ARM Architecture Reference Manual.

6.21 Inline assembler register list operands in C and C++ code

A register list can contain a maximum of 16 operands. These operands can be virtual registers or expression register operands.

The order that virtual registers and expression operands are specified in a register list is significant. The register list operands are read or written in left-to-right order. The first operand uses the lowest address, and subsequent operands use addresses formed by incrementing the previous address by four. This behavior is in contrast to the usual operation of the `LDM` or `STM` instructions where the lowest numbered physical register is always stored to the lowest memory address. This difference in behavior is a consequence of the virtualization of registers.

An expression operand or virtual register can appear more than once in a register list and is used each time it is specified.

The base register is updated, if specified. The update overwrites any value loaded into the base register during a memory load operation.

The inline assembler does not support operating on User mode registers when in a privileged mode, by specifying `^` after a register list.

Related concepts

6.20 Inline assembler expression operands in C and C++ code on page 6-236.

6.22 Inline assembler intermediate operands in C and C++ code on page 6-238.

6.22 Inline assembler intermediate operands in C and C++ code

A C or C++ constant expression of an integral type might be used as an immediate value in an inline assembly language instruction.

A constant expression that specifies an immediate shift must have a value that lies in the range defined in the *ARM Architecture Reference Manual*, as appropriate for the shift operation.

A constant expression that specifies an immediate offset for a memory or coprocessor data transfer instruction must have a value with suitable alignment.

Related concepts

6.20 Inline assembler expression operands in C and C++ code on page 6-236.

6.21 Inline assembler register list operands in C and C++ code on page 6-237.

6.23 Inline assembler function calls and branches in C and C++ code

The `BL` and `SVC` instructions of the inline assembler enable you to specify three optional lists following the normal instruction fields.

These instructions have the following format:

```
SVC{cond} svc_num, {input_param_list}, {output_value_list}, {corrupt_reg_list}
BL{cond} function, {input_param_list}, {output_value_list}, {corrupt_reg_list}
```

Note

RVCT v3.0 renamed the `SWI` instruction to `SVC`. The inline assembler still accepts `SWI` in place of `SVC`.

If you are compiling for architecture 5TE or later, the linker converts `BL` *`function`* instructions to `BLX` *`function`* instructions if appropriate. However, you cannot use `BLX` *`function`* instructions directly within inline assembly code.

- *`input_param_list`* specifies the expressions or variables that are the input parameters to the function call or `SVC` instruction, and the physical registers that contain the expressions or variables. They are specified as assignments to physical registers or as physical register names. A single list can contain both types of input register specification.

  The inline assembler ensures that the correct values are present in the specified physical registers before the `BL` or `SVC` instruction is entered. A physical register name that is specified without assignment ensures that the value in the virtual register of the same name is present in the physical register. This ensures backwards compatibility with existing inline assembly language code.

  For example, the instruction:

  ```
  BL foo, { r0=expression1, r1=expression2, r2 }
  ```

  generates the following pseudocode:

  ```
  MOV (physical) r0, expression1
  MOV (physical) r1, expression2
  MOV (physical) r2, (virtual) r2
  BL foo
  ```

  By default, if you do not specify any *`input_param_list`* input parameters, registers `r0` to `r3` are used as input parameters.

  Note

  It is not possible to specify the `lr`, `sp`, or `pc` registers in the input parameter list.

- *`output_value_list`* specifies the physical registers that contain the output values from the `BL` or `SVC` instruction and where they must be stored. The output values are specified as assignments from physical registers to modifiable lvalue expressions or as single physical register names.

  The inline assembler takes the values from the specified physical registers and assigns them into the specified expressions. A physical register name specified without assignment causes the virtual register of the same name to be updated with the value from the physical register.

  For example, the instruction:

  ```
  BL foo, { }, { result1=r0, r1 }
  ```

  generates the following pseudocode:

  ```
  BL foo
  MOV result1, (physical) r0
  MOV (virtual) r1, (physical) r1
  ```

By default, if you do not specify any *`output_value_list`* output values, register `r0` is used for the output value.

——— **Note** ———

It is not possible to specify the `lr`, `sp`, or `pc` registers in the output value list.

———————

- *`corrupt_reg_list`* specifies the physical registers that are corrupted by the called function. If the condition flags are modified by the called function, you must specify the `PSR` in the corrupted register list.

  The `BL` and `SVC` instructions always corrupt `lr`.

  If *`corrupt_reg_list`* is omitted then for `BL` and `SVC`, the registers `r0-r3`, `lr` and the `PSR` are corrupted.

  Only the branch instruction, `B`, can jump to labels within a single C or C++ function.

  By default, if you do not specify any *`corrupt_reg_list`* registers, `r0` to `r3`, `r14`, and the `PSR` can be corrupted.

  ——— **Note** ———

  It is not possible to specify the `lr`, `sp`, or `pc` registers in the corrupt register list.

  ———————

If you do not specify any lists, then:
- `r0-r3` are used as input parameters.
- `r0` is used for the output value and can be corrupted.
- `r0-r3`, `r14`, and the `PSR` can be corrupted.

——— **Note** ———

- The `BX`, `BLX`, and `BXJ` instructions are not supported in the inline assembler.
- It is not possible to specify the `lr`, `sp`, or `pc` registers in any of the input, output, or corrupted register lists.
- The `sp` register must not be changed by any `SVC` instruction or function call.

———————

6.24 Inline assembler branches and labels in C and C++ code

Labels defined in inline assembly code can be used as targets for branches or C and C++ `goto` statements.

They must be followed by a colon, `:`, like C and C++ labels, and they must be defined within the same function that they are called from.

Labels defined in C and C++ can be used as targets by branch instructions in inline assembly code, in the form:

```
B{cond} label
```

For example:

```
int foo(int x, int y)
{
  __asm
  {
     SUBS x,x,y
     BEQ end
  }
  return 1;
  end:
    return 0;
}
```

6.25 Inline assembler and virtual registers

Inline assembly code for the compiler always specifies virtual registers.

The compiler chooses the physical registers to be used for each instruction during code generation, and enables the compiler to fully optimize the assembly code and surrounding C or C++ code.

The `pc` (`r15`), `lr` (`r14`), and `sp` (`r13`) registers cannot be accessed at all. An error message is generated when these registers are accessed.

The initial values of virtual registers are undefined. Therefore, you must write to virtual registers before reading them. The compiler warns you if code reads a virtual register before writing to it. The compiler also generates these warnings for legacy code that relies on particular values in physical registers at the beginning of inline assembly code, for example:

```
int add(int i, int j)
{
    int res;
    __asm
    {
        ADD res, r0, r1   // relies on i passed in r0 and j passed in r1
    }
    return res;
}
```

This code generates warning and error messages.

The errors are generated because virtual registers `r0` and `r1` are read before writing to them. The warnings are generated because `r0` and `r1` must be defined as C or C++ variables. The corrected code is:

```
int add(int i, int j)
{
    int res;
    __asm
    {
        ADD res, i, j
    }
    return res;
}
```

Related concepts

6.14 Inline assembler and register access in C and C++ code on page 6-229.

6.26 Embedded assembler support in the compiler

The compiler enables you to include assembly code out-of-line in one or more C or C++ function definitions.

Embedded assembly code provides unrestricted, low-level access to the target processor, enables you to use the C and C++ preprocessor directives, and gives easy access to structure member offsets. The embedded assembler supports ARM and Thumb states.

Related concepts

6.27 Embedded assembler syntax in C and C++ on page 6-244.

6.28 Effect of compiler ARM and Thumb states on embedded assembler on page 6-245.

6.29 Restrictions on embedded assembly language functions in C and C++ code on page 6-246.

6.30 Compiler generation of embedded assembly language functions on page 6-247.

6.31 Access to C and C++ compile-time constant expressions from embedded assembler on page 6-249.

6.32 Differences between expressions in embedded assembler and C or C++ on page 6-250.

6.33 Manual overload resolution in embedded assembler on page 6-251.

6.34 __offsetof_base keyword for related base classes in embedded assembler on page 6-252.

6.35 Compiler-supported keywords for calling class member functions in embedded assembler on page 6-253.

6.41 Calling nonstatic member functions in embedded assembler on page 6-259.

6.42 Calling a nonvirtual member function on page 6-260.

6.43 Calling a virtual member function on page 6-261.

Related information

armasm User Guide.

6.27 Embedded assembler syntax in C and C++

An embedded assembly language function definition is marked by the **`__asm`** function qualifier in C and C++, or the **`asm`** function qualifier in C++.

The **`__asm`** and **`asm`** function qualifiers can be used on:

- Member functions.
- Non-member functions.
- Template functions.
- Template class member functions.

Functions declared with **`__asm`** or **`asm`** can have arguments, and return a type. They are called from C and C++ in the same way as normal C and C++ functions. The syntax of an embedded assembly language function is:

```
__asm return-type function-name(parameter-list)
{
  // ARM/Thumb assembly code
  instruction{;comment is optional}
  ...
  instruction
}
```

Note

Argument names are permitted in the parameter list, but they cannot be used in the body of the embedded assembly function. For example, the following function uses integer `i` in the body of the function, but this is not valid in assembly:

```
__asm int f(int i)
{
    ADD i, i, #1 // error
}
```

You can use, for example, `r0` instead of `i`.

The following example shows a string copy routine as a not very optimal embedded assembler routine.

```
#include <stdio.h>
__asm void my_strcpy(const char *src, char *dst)
{
loop
      LDRB  r2, [r0], #1
      STRB  r2, [r1], #1
      CMP   r2, #0
      BNE   loop
      BX    lr
}
int main(void)
{
    const char *a = "Hello world!";
    char b[20];
    my_strcpy (a, b);
    printf("Original string: '%s'\n", a);
    printf("Copied   string: '%s'\n", b);
    return 0;
}
```

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

6.28 Effect of compiler ARM and Thumb states on embedded assembler

The initial state of the embedded assembler, ARM or Thumb state, is determined by the initial state of the compiler, as specified on the command line.

This means that:

- If the compiler starts in ARM state, the embedded assembler uses `--arm`.
- If the compiler starts in Thumb state, the embedded assembler uses `--thumb`.

The embedded assembler state at the start of each function is as set by the invocation of the compiler, as modified by `#pragma arm` and `#pragma thumb` pragmas.

You can change the state of the embedded assembler within a function by using explicit `ARM`, `THUMB`, or `CODE16` directives in the embedded assembler function. Such a directive within an `__asm` function does not affect the ARM or Thumb state of subsequent `__asm` functions.

If you are compiling for a 32-bit Thumb capable processor, you can use both 32-bit encoded Thumb instructions and 16-bit encoded Thumb instructions when in Thumb state.

If you are compiling for a 16-bit Thumb capable processor, you can only use 16-bit encoded Thumb instructions when in Thumb state.

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

6.29 Restrictions on embedded assembly language functions in C and C++ code

A number of restrictions apply to embedded assembly language functions.

Specifically:

- After preprocessing, `__asm` functions can only contain assembly code, with the exception of the following embedded assembler built-ins:

  ```
  __cpp(expr)
  __offsetof_base(D, B)
  __mcall_is_virtual(D, f)
  __mcall_is_in_vbase(D, f)
  __mcall_offsetof_base(D, f)
  __mcall_this_offset(D, f)
  __vcall_offsetof_vfunc(D, f)
  ```
- No return instructions are generated by the compiler for an `__asm` function. If you want to return from an `__asm` function, you must include the return instructions, in assembly code, in the body of the function.

  Note

  This makes it possible to fall through to the next function, because the embedded assembler guarantees to emit the `__asm` functions in the order you define them. However, inlined and template functions behave differently. Do not assume that code execution falls out of an inline or template function into another embedded assembly function.

- `__asm` functions do not change the *ARM Architecture Procedure Call Standard* (AAPCS) rules that apply. This means that all calls between an `__asm` function and a normal C or C++ function must adhere to the AAPCS, even though there are no restrictions on the assembly code that an `__asm` function can use (for example, change state).

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

6.34 __offsetof_base keyword for related base classes in embedded assembler on page 6-252.

6.35 Compiler-supported keywords for calling class member functions in embedded assembler on page 6-253.

6.30 Compiler generation of embedded assembly language functions on page 6-247.

6.31 Access to C and C++ compile-time constant expressions from embedded assembler on page 6-249.

6.35 Compiler-supported keywords for calling class member functions in embedded assembler on page 6-253.

6.30 Compiler generation of embedded assembly language functions

The bodies of all the `__asm` functions in a translation unit are assembled as if they are concatenated into a single file that is then passed to the ARM assembler.

The order of `__asm` functions in the assembly language file that is passed to the assembler is guaranteed to be the same order as in the source file, except for functions that are generated using a template instantiation.

——— **Note** ———

This means that it is possible for control to pass from one `__asm` function to another by falling off the end of the first function into the next `__asm` function in the file, if the return instruction is omitted.

———————————

When you invoke `armcc`, the object file produced by the assembler is combined with the object file of the compiler by a partial link that produces a single object file.

The compiler generates an `AREA` directive for each `__asm` function, as in the following example:

```
#include <cstddef>
struct X
{
    int x,y;
    void addto_y(int);
};
__asm void X::addto_y(int)
{
    LDR      r2, [r0, #__cpp(offsetof(X, y))]
    ADD      r1, r2, r1
    STR      r1, [r0, #__cpp(offsetof(X, y))]
    BX       lr
}
```

For this function, the compiler generates:

```
    AREA ||.emb_text||, CODE, READONLY
    EXPORT |_ZN1X7addto_yEi|
#line num "file"
|_ZN1X7addto_yEi| PROC
    LDR r2, [r0, #4]
    ADD r1, r2, r1
    STR r1, [r0, #4]
    BX lr
    ENDP
    END
```

The use of `offsetof` must be inside `__cpp()` because it is the normal `offsetof` macro from the `cstddef` header file.

Ordinary `__asm` functions are put in an ELF section with the name `.emb_text`. That is, embedded assembly functions are never inlined. However, implicitly instantiated template functions and out-of-line copies of inline functions are placed in an area with a name that is derived from the name of the function, and an extra attribute that marks them as common. This ensures that the special semantics of these kinds of functions are maintained.

——— **Note** ———

Because of the special naming of the area for out-of-line copies of inline functions and template functions, these functions are not in the order of definition, but in an arbitrary order. Therefore, do not assume that code execution falls out of an inline or template function and into another `__asm` function.

———————————

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

6.29 Restrictions on embedded assembly language functions in C and C++ code on page 6-246.

Related information

ELF for the ARM Architecture.

6.31 Access to C and C++ compile-time constant expressions from embedded assembler

You can use the **`__cpp`** keyword to access C and C++ compile-time constant expressions, including the addresses of data or functions with external linkage, from the assembly code.

The expression inside the `__cpp` must be a constant expression suitable for use as a C++ static initialization. See *3.6.2 Initialization of non-local objects* and *5.19 Constant expressions* in ISO/IEC 14882:2003.

The following example shows a constant replacing the use of `__cpp(`*`expr`*`)`:

```
LDR r0, =__cpp(&some_variable)
LDR r1, =__cpp(some_function)
BL  __cpp(some_function)
MOV r0, #__cpp(some_constant_expr)
```

Names in the `__cpp` expression are looked up in the C++ context of the `__asm` function. Any names in the result of a `__cpp` expression are mangled as required and automatically have `IMPORT` statements generated for them.

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.
6.29 Restrictions on embedded assembly language functions in C and C++ code on page 6-246.
6.33 Manual overload resolution in embedded assembler on page 6-251.
6.32 Differences between expressions in embedded assembler and C or C++ on page 6-250.

6.32 Differences between expressions in embedded assembler and C or C++

There are a number of differences between embedded assembly and C or C++.

Specifically:

- Assembly expressions are always unsigned. The same expression might have different values between assembly and C or C++. For example:

```
MOV r0, #(-33554432 / 2)        // result is 0x7f000000
MOV r0, #__cpp(-33554432 / 2)   // result is 0xff000000
```

- Assembly numbers with leading zeros are still decimal. For example:

```
MOV r0, #0700                   // decimal 700
MOV r0, #__cpp(0700)            // octal 0700 == decimal 448
```

- Assembly operator precedence differs from C and C++. For example:

```
MOV r0, #(0x23 :AND: 0xf + 1)      // ((0x23 & 0xf) + 1) => 4
MOV r0, #__cpp(0x23 & 0xf + 1)      // (0x23 & (0xf + 1)) => 0
```

- Assembly strings are not `NUL`-terminated:

```
DCB "Hello world!"                 // 12 bytes (no trailing NUL)
DCB __cpp("Hello world!")          // 13 bytes (trailing NUL)
```

Note

The embedded assembly rules apply outside `__cpp`, and the C or C++ rules apply inside `__cpp`.

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

6.31 Access to C and C++ compile-time constant expressions from embedded assembler on page 6-249.

6.33 Manual overload resolution in embedded assembler

The following example shows the use of C++ casts to do overload resolution for nonvirtual function calls.

```
void g(int);
void g(long);
struct T
{
    int mf(int);
    int mf(int,int);
};
__asm void  f(T*, int, int)
{
    BL __cpp(static_cast<int (T::*)(int, int)>(&T::mf)) // calls T::mf(int, int)
    BL __cpp(static_cast<void (*)(int)>(g)) // calls g(int)
    BX lr
}
```

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

6.31 Access to C and C++ compile-time constant expressions from embedded assembler on page 6-249.

6.34 __offsetof_base keyword for related base classes in embedded assembler

The `__offsetof_base` keyword enables you to determine the offset from the beginning of an object to a base class sub-object within it.

```
__offsetof_base(D, B)
```

`B` must be an unambiguous, nonvirtual base class of `D`.

Returns the offset from the beginning of a `D` object to the start of the `B` base subobject within it. The result might be zero. The following example shows the offset (in bytes) that must be added to a `D* p` to implement the equivalent of `static_cast<B*>(p)`.

```
__asm B* my_static_base_cast(D* /*p*/) // equivalent to:
                                       // return static_cast<B*>(p)
{
    if __offsetof_base(D, B) <> 0 // optimize zero offset case
        ADD r0, r0, #__offsetof_base(D, B)
    endif
    BX lr
}
```

The **`__offsetof_base`**, **`__mcall_`***, and **`_vcall_offsetof_vfunc`** keywords are converted into integer or logical constants in the assembly source code. You can only use it in `__asm` functions, not in `__cpp` expressions.

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

6.29 Restrictions on embedded assembly language functions in C and C++ code on page 6-246.

6.35 Compiler-supported keywords for calling class member functions in embedded assembler

The following embedded assembler built-ins facilitate the calling of virtual and nonvirtual member functions from an `__asm` function.

Those beginning with `__mcall` can be used for both virtual and nonvirtual functions. Those beginning with `__vcall` can be used only with virtual functions. They do not particularly help in calling static member functions.

- `__mcall_is_virtual(D, f)`.
- `__mcall_is_in_vbase(D, f)`.
- `__mcall_offsetof_vbase(D, f)`.
- `__mcall_this_offset(D, f)`.
- `__vcall_offsetof_vfunc(D, f)`.

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.
6.36 __mcall_is_virtual(D, f) on page 6-254.
6.37 __mcall_is_in_vbase(D, f) on page 6-255.
6.38 __mcall_offsetof_vbase(D, f) on page 6-256.
6.39 __mcall_this_offset(D, f) on page 6-257.
6.40 __vcall_offsetof_vfunc(D, f) on page 6-258.
6.41 Calling nonstatic member functions in embedded assembler on page 6-259.
6.29 Restrictions on embedded assembly language functions in C and C++ code on page 6-246.

6.36 __mcall_is_virtual(D, f)

Results in `{TRUE}` if `f` is a virtual member function found in `D`, or a base class of `D`, otherwise `{FALSE}`.

If it returns `{TRUE}` the call can be done using virtual dispatch, otherwise the call must be done directly.

Related concepts

6.35 Compiler-supported keywords for calling class member functions in embedded assembler on page 6-253.

6.42 Calling a nonvirtual member function on page 6-260.

6.43 Calling a virtual member function on page 6-261.

6.37 __mcall_is_in_vbase(D, f)

Results in `{TRUE}` if `f` is a nonstatic member function found in a virtual base class of `D`, otherwise `{FALSE}`.

If it returns `{TRUE}` the `this` adjustment must be done using `__mcall_offsetof_vbase(D, f)`, otherwise it must be done with `__mcall_this_offset(D, f)`.

Related concepts

6.35 Compiler-supported keywords for calling class member functions in embedded assembler on page 6-253.

6.42 Calling a nonvirtual member function on page 6-260.

6.43 Calling a virtual member function on page 6-261.

6.38 __mcall_offsetof_vbase(D, f)

Returns the negative offset from the value of the vtable pointer of the vtable slot that holds the base offset (from the beginning of a `D` object to the start of the base that `f` is defined in).

Where `D` is a class type and `f` is a nonstatic member function defined in a virtual base class of `D`, in other words `__mcall_is_in_vbase(D,f)` returns `{TRUE}`.

The base offset is the `this` adjustment necessary when making a call to `f` with a pointer to a `D`.

———— **Note** ————

The offset returns a positive number that then has to be subtracted from the value of the vtable pointer.

————————————

Related concepts

6.35 Compiler-supported keywords for calling class member functions in embedded assembler on page 6-253.

6.42 Calling a nonvirtual member function on page 6-260.

6.43 Calling a virtual member function on page 6-261.

6.39 __mcall_this_offset(D, f)

Returns the offset from the beginning of a `D` object to the start of the base in which `f` is defined.

This is the `this` adjustment necessary when making a call to `f` with a pointer to a `D`. It is either zero if `f` is found in `D` or the same as `__offsetof_base(D,B)`, where `B` is a nonvirtual base class of `D` that contains `f`.

Where `D` is a class type and `f` is a nonstatic member function defined in `D` or a nonvirtual base class of `D`.

If `__mcall_this_offset(D,f)` is used when `f` is found in a virtual base class of `D` it returns an arbitrary value designed to cause an assembly error if used. This is so that such invalid uses of `__mcall_this_offset` can occur in sections of assembly code that are to be skipped.

Related concepts

6.35 Compiler-supported keywords for calling class member functions in embedded assembler on page 6-253.

6.42 Calling a nonvirtual member function on page 6-260.

6.43 Calling a virtual member function on page 6-261.

6.40 __vcall_offsetof_vfunc(D, f)

Returns the offset of the slot in the vtable that holds the pointer to the virtual function, `f`.

Where `D` is a class and `f` is a virtual function defined in `D`, or a base class of `D`.

If `__vcall_offsetof_vfunc(D, f)` is used when `f` is not a virtual member function it returns an arbitrary value designed to cause an assembly error if used.

Related concepts

6.35 Compiler-supported keywords for calling class member functions in embedded assembler on page 6-253.

6.42 Calling a nonvirtual member function on page 6-260.

6.43 Calling a virtual member function on page 6-261.

6.41 Calling nonstatic member functions in embedded assembler

You can use keywords beginning with `__mcall` and `__vcall` to call nonvirtual and virtual functions from `__asm` functions.

There is no `__mcall_is_static` to detect static member functions because static member functions have different parameters (that is, no `this`), so call sites are likely to already be specific to calling a static member function.

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

6.35 Compiler-supported keywords for calling class member functions in embedded assembler on page 6-253.

6.42 Calling a nonvirtual member function on page 6-260.

6.43 Calling a virtual member function on page 6-261.

6.42 Calling a nonvirtual member function

The following example shows code that calls a nonvirtual function in either a virtual or nonvirtual base.

```
// rp contains a D* and we want to do the equivalent of rp->f() where f is a
// nonvirtual function
// all arguments other than the this pointer are already set up
// assumes f does not return a struct
 if __mcall_is_in_vbase(D, f)
   LDR r12, [rp]                                // fetch vtable pointer
   LDR r0, [r12, #-__mcall_offsetof_vbase(D, f)]   // fetch the vbase offset
   ADD r0, r0, rp                               // do this adjustment
 else
   ADD r0, rp, #__mcall_this_offset(D, f)          // set up and adjust this
                                                // pointer for D*
 endif
   BL __cpp(&D::f)                              // call D::f
```

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

6.41 Calling nonstatic member functions in embedded assembler on page 6-259.

6.43 Calling a virtual member function on page 6-261.

6.43 Calling a virtual member function

The following example shows code that calls a virtual function in either a virtual or nonvirtual base.

```
// rp contains a D* and we want to do the equivalent of rp->f() where f is a
// virtual function
// all arguments other than the this pointer are already set up
// assumes f does not return a struct
 if __mcall_is_in_vbase(D, f)
    LDR r12, [rp]                                 // fetch vtable pointer
    LDR r0, [r12, #-__mcall_offsetof_vbase(D, f)] // fetch the base offset
    ADD r0, r0, rp                                // do this adjustment
    LDR r12, [r0]                                 // fetch vbase vtable pointer
 else
    MOV r0, rp                                    // set up this pointer for D*
    LDR r12, [rp]                                 // fetch vtable pointer
    ADD r0, r0, #__mcall_this_offset(D, f)        // do this adjustment
 endif
    MOV lr, pc                                    // prepare lr
    LDR pc, [r12, #__vcall_offsetof_vfunc(D, f)]  // calls rp->f()
```

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

6.41 Calling nonstatic member functions in embedded assembler on page 6-259.

6.42 Calling a nonvirtual member function on page 6-260.

6.44 Accessing sp (r13), lr (r14), and pc (r15)

The following methods enable you to access the `sp`, `lr`, and `pc` registers correctly in your source code.

The first method uses the compiler intrinsics in inline assembly, for example:

```
void printReg()
{
    unsigned int spReg, lrReg, pcReg;
    __asm
    {
        MOV spReg, __current_sp()
        MOV pcReg, __current_pc()
        MOV lrReg, __return_address()
    }
    printf("SP = 0x%X\n",spReg);
    printf("PC = 0x%X\n",pcReg);
    printf("LR = 0x%X\n",lrReg);
}
```

The second method uses embedded assembly to access physical ARM registers from within a C or C++ source file, for example:

```
__asm void func()
{
    MOV r0, lr
    ...
    BX lr
}
```

This enables the return address of a function to be captured and displayed, for example, for debugging purposes, to show the call tree.

Note

The compiler might also inline a function into its caller function. If a function is inlined, then the return address is the return address of the function that calls the inlined function. Also, a function might be tail called.

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

Related references

9.134 __return_address intrinsic on page 9-659.

9.108 __current_pc intrinsic on page 9-630.

9.109 __current_sp intrinsic on page 9-631.

6.45 Differences in compiler support for inline and embedded assembly code

There are differences between the ways inline and embedded assembly are compiled.

Specifically:

- Inline assembly code uses a high level of processor abstraction, and is integrated with the C and C++ code during code generation. Therefore, the compiler optimizes the C and C++ code and the assembly code together.
- Unlike inline assembly code, embedded assembly code is assembled separately from the C and C++ code to produce a compiled object that is then combined with the object from the compilation of the C or C++ source.
- Inline assembly code can be inlined by the compiler, but embedded assembly code cannot be inlined, either implicitly or explicitly.

The following table summarizes the main differences between inline assembler and embedded assembler.

Table 6-1 Differences between inline and embedded assembler

Feature	Embedded assembler	Inline assembler
Instruction set	ARM and Thumb.	ARM on all processors. Thumb on processors with Thumb-2 technology.
ARM assembler directives	All supported.	None supported.
ARMv6 instructions	All supported.	Supports most instructions, with some exceptions, for example `SETEND` and some of the system extensions. The complete set of ARMv6 SIMD instructions is supported.
ARMv7 instructions	All supported.	Supports most instructions.
VFP instructions	All supported.	VFPv2 only.
C/C++ expressions	Constant expressions only.	Full C/C++ expressions.
Optimization of assembly code	No optimization.	Full optimization.
Inlining	Never.	Possible.
Register access	Specified physical registers are used. You can also use `PC`, `LR` and `SP`.	Uses virtual registers. Using `sp` (`r13`), `lr` (`r14`), and `pc` (`r15`) gives an error.
Return instructions	You must add them in your code.	Generated automatically. (The `BX`, `BXJ`, and `BLX` instructions are not supported.)
`BKPT` instruction	Supported directly.	Not supported.

Chapter 7
Compiler Command-line Options

Describes the `armcc` compiler command-line options.

It contains the following sections:

- *7.1 -Aopt* on page 7-268.
- *7.2 --allow_fpreg_for_nonfpdata, --no_allow_fpreg_for_nonfpdata* on page 7-269.
- *7.3 --allow_null_this, --no_allow_null_this* on page 7-270.
- *7.4 --alternative_tokens, --no_alternative_tokens* on page 7-271.
- *7.5 --anachronisms, --no_anachronisms* on page 7-272.
- *7.6 --apcs=qualifier...qualifier* on page 7-273.
- *7.7 --arm* on page 7-277.
- *7.8 --arm_only* on page 7-278.
- *7.9 --asm* on page 7-279.
- *7.10 --asm_dir=directory_name* on page 7-280.
- *7.11 --autoinline, --no_autoinline* on page 7-281.
- *7.12 --bigend* on page 7-282.
- *7.13 --bitband* on page 7-283.
- *7.14 --branch_tables, --no_branch_tables* on page 7-284.
- *7.15 --brief_diagnostics, --no_brief_diagnostics* on page 7-286.
- *7.16 --bss_threshold=num* on page 7-287.
- *7.17 -c* on page 7-288.
- *7.18 -C* on page 7-289.
- *7.19 --c90* on page 7-290.
- *7.20 --c99* on page 7-291.
- *7.21 --code_gen, --no_code_gen* on page 7-292.
- *7.22 --compatible=name* on page 7-293.
- *7.23 --compile_all_input, --no_compile_all_input* on page 7-295.

- *7.24 --conditionalize, --no_conditionalize* on page 7-296.
- *7.25 --cpp* on page 7-297.
- *7.26 --cpp11* on page 7-298.
- *7.27 --cpp_compat* on page 7-299.
- *7.28 --cpu=list* on page 7-301.
- *7.29 --cpu=name compiler option* on page 7-302.
- *7.30 --create_pch=filename* on page 7-304.
- *7.31 -Dname[(parm-list)][=def]* on page 7-305.
- *7.32 --data_reorder, --no_data_reorder* on page 7-306.
- *7.33 --debug, --no_debug* on page 7-307.
- *7.34 --debug_macros, --no_debug_macros* on page 7-308.
- *7.35 --default_extension=ext* on page 7-309.
- *7.36 --dep_name, --no_dep_name* on page 7-310.
- *7.37 --depend=filename* on page 7-311.
- *7.38 --depend_dir=directory_name* on page 7-312.
- *7.39 --depend_format=string* on page 7-313.
- *7.40 --depend_single_line, --no_depend_single_line* on page 7-314.
- *7.41 --depend_system_headers, --no_depend_system_headers* on page 7-315.
- *7.42 --depend_target=target* on page 7-316.
- *7.43 --diag_error=tag[,tag,...]* on page 7-317.
- *7.44 --diag_remark=tag[,tag,...]* on page 7-318.
- *7.45 --diag_style=arm|ide|gnu compiler option* on page 7-319.
- *7.46 --diag_suppress=tag[,tag,...]* on page 7-320.
- *7.47 --diag_suppress=optimizations* on page 7-321.
- *7.48 --diag_warning=tag[,tag,...]* on page 7-322.
- *7.49 --diag_warning=optimizations* on page 7-323.
- *7.50 --dollar, --no_dollar* on page 7-324.
- *7.51 --dwarf2* on page 7-325.
- *7.52 --dwarf3* on page 7-326.
- *7.53 -E* on page 7-327.
- *7.54 --echo* on page 7-328.
- *7.55 --emit_frame_directives, --no_emit_frame_directives* on page 7-329.
- *7.56 --enum_is_int* on page 7-330.
- *7.57 --errors=filename* on page 7-331.
- *7.58 --exceptions, --no_exceptions* on page 7-332.
- *7.59 --exceptions_unwind, --no_exceptions_unwind* on page 7-333.
- *7.60 --execute_only* on page 7-334.
- *7.61 --extended_initializers, --no_extended_initializers* on page 7-335.
- *7.62 --feedback=filename* on page 7-336.
- *7.63 --float_literal_pools, --no_float_literal_pools* on page 7-337.
- *7.64 --force_new_nothrow, --no_force_new_nothrow* on page 7-338.
- *7.65 --forceinline* on page 7-339.
- *7.66 --fp16_format=format* on page 7-340.
- *7.67 --fpmode=model* on page 7-341.
- *7.68 --fpu=list* on page 7-343.
- *7.69 --fpu=name compiler option* on page 7-344.
- *7.70 --friend_injection, --no_friend_injection* on page 7-347.
- *7.71 -g* on page 7-348.
- *7.72 --global_reg=reg_name[,reg_name,...]* on page 7-349.
- *7.73 --gnu* on page 7-350.
- *7.74 --gnu_defaults* on page 7-351.
- *7.75 --gnu_instrument, --no_gnu_instrument* on page 7-352.
- *7.76 --gnu_version=version* on page 7-353.
- *7.77 --guiding_decls, --no_guiding_decls* on page 7-354.
- *7.78 --help* on page 7-355.
- *7.79 -Idir[,dir,...]* on page 7-356.

- *7.80 --ignore_missing_headers* on page 7-357.
- *7.81 --implicit_include, --no_implicit_include* on page 7-358.
- *7.82 --implicit_include_searches, --no_implicit_include_searches* on page 7-359.
- *7.83 --implicit_key_function, --no_implicit_key_function* on page 7-360.
- *7.84 --implicit_typename, --no_implicit_typename* on page 7-361.
- *7.85 --info=totals* on page 7-362.
- *7.86 --inline, --no_inline* on page 7-363.
- *7.87 --integer_literal_pools, --no_integer_literal_pools* on page 7-364.
- *7.88 --interface_enums_are_32_bit* on page 7-365.
- *7.89 --interleave* on page 7-366.
- *7.90 -Jdir[,dir,...]* on page 7-367.
- *7.91 --kandr_include* on page 7-368.
- *7.92 -Lopt* on page 7-369.
- *7.93 --library_interface=lib* on page 7-370.
- *7.94 --library_type=lib* on page 7-372.
- *7.95 --liclinger=seconds* on page 7-373.
- *7.96 --licretry* on page 7-374.
- *7.97 --link_all_input, --no_link_all_input* on page 7-375.
- *7.98 --list* on page 7-376.
- *7.99 --list_dir=directory_name* on page 7-378.
- *7.100 --list_macros* on page 7-379.
- *7.101 --littleend* on page 7-380.
- *7.102 --locale=lang_country* on page 7-381.
- *7.103 --long_long* on page 7-382.
- *7.104 --loop_optimization_level=opt* on page 7-383.
- *7.105 --loose_implicit_cast* on page 7-384.
- *7.106 --lower_ropi, --no_lower_ropi* on page 7-385.
- *7.107 --lower_rwpi, --no_lower_rwpi* on page 7-386.
- *7.108 -M* on page 7-387.
- *7.109 --md* on page 7-388.
- *7.110 --message_locale=lang_country[.codepage]* on page 7-389.
- *7.111 --min_array_alignment=opt* on page 7-390.
- *7.112 --mm* on page 7-391.
- *7.113 --multibyte_chars, --no_multibyte_chars* on page 7-392.
- *7.114 --multifile, --no_multifile* on page 7-393.
- *7.115 --multiply_latency=cycles* on page 7-394.
- *7.116 --narrow_volatile_bitfields* on page 7-395.
- *7.117 --nonstd_qualifier_deduction, --no_nonstd_qualifier_deduction* on page 7-396.
- *7.118 -o filename* on page 7-397.
- *7.119 -Onum* on page 7-399.
- *7.120 --old_specializations, --no_old_specializations* on page 7-402.
- *7.121 --old_style_preprocessing* on page 7-403.
- *7.122 --omf_browse* on page 7-404.
- *7.123 --ool_section_name, --no_ool_section_name* on page 7-405.
- *7.124 -Ospace* on page 7-406.
- *7.125 -Otime* on page 7-407.
- *7.126 --output_dir=directory_name* on page 7-408.
- *7.127 -P* on page 7-409.
- *7.128 --parse_templates, --no_parse_templates* on page 7-410.
- *7.129 --pch* on page 7-411.
- *7.130 --pch_dir=dir* on page 7-412.
- *7.131 --pch_messages, --no_pch_messages* on page 7-413.
- *7.132 --pch_verbose, --no_pch_verbose* on page 7-414.
- *7.133 --pending_instantiations=n* on page 7-415.
- *7.134 --phony_targets* on page 7-416.
- *7.135 --pointer_alignment=num* on page 7-417.

- *7.136 --preinclude=filename* on page 7-418.
- *7.137 --preprocess_assembly* on page 7-419.
- *7.138 --preprocessed* on page 7-420.
- *7.139 --protect_stack, --no_protect_stack* on page 7-421.
- *7.140 --reassociate_saturation, --no_reassociate_saturation* on page 7-422.
- *7.141 --reduce_paths, --no_reduce_paths* on page 7-423.
- *7.142 --relaxed_ref_def, --no_relaxed_ref_def* on page 7-424.
- *7.143 --remarks* on page 7-425.
- *7.144 --remove_unneeded_entities, --no_remove_unneeded_entities* on page 7-426.
- *7.145 --restrict, --no_restrict* on page 7-427.
- *7.146 --retain=option* on page 7-428.
- *7.147 --rtti, --no_rtti* on page 7-429.
- *7.148 --rtti_data, --no_rtti_data* on page 7-430.
- *7.149 -S* on page 7-431.
- *7.150 --share_inlineable_strings, --no_share_inlineable_strings* on page 7-432.
- *7.151 --show_cmdline* on page 7-434.
- *7.152 --signed_bitfields, --unsigned_bitfields* on page 7-435.
- *7.153 --signed_chars, --unsigned_chars* on page 7-436.
- *7.154 --split_ldm* on page 7-437.
- *7.155 --split_sections* on page 7-438.
- *7.156 --strict, --no_strict* on page 7-439.
- *7.157 --strict_warnings* on page 7-440.
- *7.158 --string_literal_pools, --no_string_literal_pools* on page 7-441.
- *7.159 --sys_include* on page 7-443.
- *7.160 --thumb* on page 7-444.
- *7.161 --trigraphs, --no_trigraphs* on page 7-445.
- *7.162 --type_traits_helpers, --no_type_traits_helpers* on page 7-446.
- *7.163 -Uname* on page 7-447.
- *7.164 --unaligned_access, --no_unaligned_access* on page 7-448.
- *7.165 --use_frame_pointer, --no_use_frame_pointer* on page 7-450.
- *7.166 --use_pch=filename* on page 7-451.
- *7.167 --using_std, --no_using_std* on page 7-452.
- *7.168 --version_number* on page 7-453.
- *7.169 --vfe, --no_vfe* on page 7-454.
- *7.170 --via=filename* on page 7-455.
- *7.171 --vla, --no_vla* on page 7-456.
- *7.172 --vsn* on page 7-457.
- *7.173 -W* on page 7-458.
- *7.174 --wchar, --no_wchar* on page 7-459.
- *7.175 --wchar16* on page 7-460.
- *7.176 --wchar32* on page 7-461.
- *7.177 --whole_program* on page 7-462.
- *7.178 --wrap_diagnostics, --no_wrap_diagnostics* on page 7-463.

7.1 -Aopt

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.

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 assembly code.

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

Restrictions

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

Example

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

Related references

7.92 -Lopt on page 7-369.
7.151 --show_cmdline on page 7-434.
7.7 --arm on page 7-277.
7.22 --compatible=name on page 7-293.
7.28 --cpu=list on page 7-301.
7.29 --cpu=name compiler option on page 7-302.

7.2 --allow_fpreg_for_nonfpdata, --no_allow_fpreg_for_nonfpdata

Enables and disables the use of VFP registers and data transfer instructions for non-VFP data.

Usage

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

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

Default

The default is `--no_allow_fpreg_for_nonfpdata`.

Related references

7.67 --fpmode=model on page 7-341.
7.68 --fpu=list on page 7-343.
7.69 --fpu=name compiler option on page 7-344.

Related information

Extension register bank mapping.
VFP views of the extension register bank.

7.3 --allow_null_this, --no_allow_null_this

Allows and disallows null **`this`** pointers in C++.

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.

Default

The default is `--no_allow_null_this`.

Related references

7.74 --gnu_defaults on page 7-351.

7.4 --alternative_tokens, --no_alternative_tokens

Enables and disables the recognition of alternative tokens in C and C++.

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`**.

Default

The default is `--alternative_tokens`.

7.5 --anachronisms, --no_anachronisms

Enables and disables anachronisms in C++.

Mode

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

Default

The default is `--no_anachronisms`.

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.

Related references

7.25 --cpp on page 7-297.

7.156 --strict, --no_strict on page 7-439.

7.157 --strict_warnings on page 7-440.

10.8 Anachronisms in ARM C++ on page 10-718.

7.6 --apcs=qualifier...qualifier

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.

Syntax

`--apcs=`*`qualifier...qualifier`*

Where *`qualifier...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 ARM recommends that you use `--apcs` instead.

Note

The `/` prefix is optional for the first qualifier, but must be present to separate subsequent qualifiers in the same `--apcs` option. For example, `--apcs=/nointerwork/noropi/norwpi` is equivalent to `--apcs=nointerwork/noropi/norwpi`.

You can specify multiple qualifiers using either a single `--apcs` option or multiple `--apcs` options. For example, `--apcs=/nointerwork/noropi/norwpi` is equivalent to `--apcs=/nointerwork --apcs=noropi/norwpi`.

Default

If you do not specify an `--apcs` option, the compiler assumes `--apcs=/nointerwork/noropi/norwpi/nofpic`.

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++.

`/rwpi`
`/norwpi`

If you select the `/rwpi` 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_rwpi` 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 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.

`/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:

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

- 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.

`/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:

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

- 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.

Restrictions

There are restrictions when you compile code with `/ropi`, or `/rwpi`, 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
```

`/rwpi`

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

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

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

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

```
armcc --apcs=/rwpi
```

  Note

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

Related concepts

4.51 Compiler options for floating-point linkage and computations on page 4-167.
4.42 Default selection of hardware or software floating-point support on page 4-156.

Related references

7.69 --fpu=name compiler option on page 7-344.
7.106 --lower_ropi, --no_lower_ropi on page 7-385.
7.107 --lower_rwpi, --no_lower_rwpi on page 7-386.
__declspec(dllexport).

Related information

Procedure Call Standard for the ARM Architecture.
ARM C libraries and multithreading.
Overview of veneers.

7.7 --arm

Targets 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.

Default

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

Related references

7.8 --arm_only on page 7-278.
7.160 --thumb on page 7-444.
9.76 #pragma arm on page 9-595.
7.1 -Aopt on page 7-268.
7.22 --compatible=name on page 7-293.
7.28 --cpu=list on page 7-301.
7.29 --cpu=name compiler option on page 7-302.

Related information

ARM architectures supported by the toolchain.

7.8 --arm_only

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.

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`.

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.

Related references

7.7 --arm on page 7-277.

7.160 --thumb on page 7-444.

Related information

--16 assembler option.

--32 assembler option.

Order of options on the command line.

7.9 --asm

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`.

Usage

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

Table 7-1 Compiling with the --asm option

Compiler option	Action
`--asm`	Writes a listing to a file of the disassembly of the compiled source. The link step is also performed, unless the `-c` 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 `.s`.
`--asm -c`	As for `--asm`, except that the link step is not performed.
`--asm --interleave`	As for `--asm`, 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 `.txt`.
`--asm --multifile`	As for `--asm`, except that the compiler produces empty object files for the files merged into the main file.
`--asm -o` *`filename`*	As for `--asm`, except that the object file is named *`filename`*. The disassembly is written to the file *`filename.s`*. The name of the object file must not have the filename extension `.s`. If the filename extension of the object file is `.s`, the disassembly is written over the top of the object file.

Related references

7.17 -c on page 7-288.
7.89 --interleave on page 7-366.
7.114 --multifile, --no_multifile on page 7-393.
7.118 -o filename on page 7-397.
7.149 -S on page 7-431.
2.7 Filename suffixes recognized by the compiler on page 2-47.

7.10 --asm_dir=directory_name

Specifies a directory for disassembly output files created by the `--asm` and `-S` options.

Default

If the `--asm_dir` option is not used, disassembly 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).

Note

The `--asm_dir` option has no effect unless you also specify either the `--asm` or the `-S` options.

Example

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

Result:

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

Related references

7.9 --asm on page 7-279.
7.38 --depend_dir=directory_name on page 7-312.
7.99 --list_dir=directory_name on page 7-378.
7.126 --output_dir=directory_name on page 7-408.

7.11 --autoinline, --no_autoinline

Enables and disables 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.

Usage

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

Default

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

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

Related concepts

3.32 Default compiler options that are affected by optimization level on page 3-102.

Related references

7.65 --forceinline on page 7-339.
7.86 --inline, --no_inline on page 7-363.
7.119 -Onum on page 7-399.
7.124 -Ospace on page 7-406.
7.125 -Otime on page 7-407.

7.12 --bigend

Generates code suitable 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.

Default

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

Related references

7.101 --littleend on page 7-380.

Related information

--be8 linker option.

--be32 linker option.

7.13 --bitband

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.

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.

Example

In this example, the writes to bitfields `i` and `k` are bit-banded when compiled using the `--bitband` command-line option.

```
typedef struct {
  int i : 1;
  int j : 2;
  int k : 1;
} BB;
BB value;
void update_value(void)
{
  value.i = 1;
  value.k = 1;
}
```

Related concepts

3.14 Compiler and processor support for bit-banding on page 3-81.

7.14 --branch_tables, --no_branch_tables

Controls whether the compiler places branch tables for switch statements in the code section or a separate data section.

The compiler uses several different techniques to generate code for switch statements. Some of these techniques create a table of branch offsets.

With the `--branch_tables` option, the compiler places the branch offset table in the code section. In the following example, lines highlighted with *** contain these branch offsets:

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 72 bytes (alignment 2)
    Address: 0x00000000

    $t
    .text
    f
        0x00000000:    b510        ..      PUSH     {r4,lr}
        0x00000002:    2807        .(      CMP      r0,#7
        0x00000004:    d21b        ..      BCS      {pc}+0x3a ; 0x3e
        0x00000006:    e8dff000    ....    TBB      [pc,r0]
    $d
        0x0000000a:    0704        ..      DCW    1796                     ***
        0x0000000c:    13100d0a    ....    DCDU   319819018                ***
        0x00000010:    0016        ..      DCW    22                       ***
    $t
        0x00000012:    2005        .       MOVS     r0,#5
        0x00000014:    f7fffffe    ....    BL       g
```

The `--no_branch_tables` option instructs the compiler to insert the branch offset table into a separate data section instead:

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 72 bytes (alignment 4)
    Address: 0x00000000

    $t
    .text
    f
        0x00000000:    b510        ..      PUSH     {r4,lr}
        0x00000002:    2807        .(      CMP      r0,#7
        0x00000004:    d218        ..      BCS      {pc}+0x34 ; 0x38
        0x00000006:    4b0f        .K      LDR      r3,[pc,#60] ; [0x44] = 0
        0x00000008:    e8d3f000    ....    TBB      [r3,r0]
        0x0000000c:    2005        .       MOVS     r0,#5
        0x0000000e:    f7fffffe    ....    BL       g
        ...

** Section #4 'c.f.00000006' (SHT_PROGBITS) [SHF_ALLOC]
    Size   : 7 bytes
    Address: 0x00000000

    0x000000:   00 03 06 09 0c 0f 12                               .......
```

Default

The default is `--branch_tables`.

`--execute_only` implies `--no_branch_tables`, unless `--branch_tables` is explicitly specified.

Note

Do not use `--execute_only` in conjunction with `--branch_tables`. If you do, then the compiler places the branch offset table in an unreadable, execute-only code region.

Related concepts

3.19 Compiler support for literal pools on page 3-86.

Related references

7.87 --integer_literal_pools, --no_integer_literal_pools on page 7-364.

7.158 --string_literal_pools, --no_string_literal_pools on page 7-441.

7.63 --float_literal_pools, --no_float_literal_pools on page 7-337.

7.60 --execute_only on page 7-334.

7.15 --brief_diagnostics, --no_brief_diagnostics

Enables and disables the output of brief diagnostic messages.

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.

Default

The default is `--no_brief_diagnostics`.

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 ";"
```

Related references

7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

7.16 --bss_threshold=num

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.

Syntax

`--bss_threshold=`*`num`*

Where:

`num`

is either:

`0`

place small global ZI data items in ZI data sections

`8`

place small global ZI data items in RW data sections.

Usage

In RVCT 2.1 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))`.

———————————

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`.

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.

Related references

9.77 #pragma arm section [section_type_list] on page 9-596.
9.67 __attribute__((section("name"))) variable attribute on page 9-586.
9.73 __attribute__((zero_init)) variable attribute on page 9-592.

7.17 -c

Instructs the compiler to perform the compilation step, but not the link step.

——— **Note** ———

This option is different from the uppercase -C option.

———————————

Usage

ARM recommends using the `-c` option in projects with more than one source file.

Related references

7.9 --asm on page 7-279.

7.98 --list on page 7-376.

7.118 -o filename on page 7-397.

7.149 -S on page 7-431.

7.18 -C

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.

Related references

7.53 -E on page 7-327.

7.19 --c90

Enables the compilation of C90 source code.

It enforces C only, and C++ syntax is not accepted.

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.

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.

Related references

7.20 --c99 on page 7-291.
7.73 --gnu on page 7-350.
7.25 --cpp on page 7-297.
7.156 --strict, --no_strict on page 7-439.
7.26 --cpp11 on page 7-298.
7.27 --cpp_compat on page 7-299.
1.2 Source language modes of the compiler on page 1-29.
2.7 Filename suffixes recognized by the compiler on page 2-47.

7.20 --c99

Enables the compilation of C99 source code.

It enforces C only, and C++ syntax is not accepted.

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.

Default

For files having a suffix of `.c`, `.ac`, or `.tc`, `--c90` applies by default.

Related references

7.19 --c90 on page 7-290.
7.73 --gnu on page 7-350.
7.25 --cpp on page 7-297.
7.156 --strict, --no_strict on page 7-439.
7.26 --cpp11 on page 7-298.
7.27 --cpp_compat on page 7-299.
1.2 Source language modes of the compiler on page 1-29.
2.7 Filename suffixes recognized by the compiler on page 2-47.

7.21 --code_gen, --no_code_gen

Enables and disables the generation of object code.

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

Default

The default is `--code_gen`.

7.22 --compatible=name

Generates code that is compatible with multiple target processors.

Syntax

```
--compatible=name
```

Where:

name

is the name of a target processor or `None`.

Processor names are not case-sensitive.

Specifying `None` generates code only for the processor specified by `--cpu`.

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.

Default

The default is `None`.

Usage

Using this option avoids having to recompile the same source code for different targets.

See the following table. The valid combinations are:

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

Table 7-2 Compatible processor or architecture combinations

Group 1	`ARM7TDMI`
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 generate 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.

Example

To generate code that is compatible with both the ARM7TDMI processor and the Cortex-M4 processor, specify:

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

Related references

7.1 -Aopt on page 7-268.

7.7 --arm on page 7-277.

7.28 --cpu=list on page 7-301.

7.29 --cpu=name compiler option on page 7-302.

7.23 --compile_all_input, --no_compile_all_input

Enables and disables 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`.

Default

The default is `--no_compile_all_input`.

Related references

7.97 --link_all_input, --no_link_all_input on page 7-375.

2.7 Filename suffixes recognized by the compiler on page 2-47.

7.24 --conditionalize, --no_conditionalize

Enables and disables 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`.

Default

The default is `--conditionalize`.

Related information

Conditional instructions.
Condition code suffixes.
Comparison of condition code meanings.

7.25 --cpp

Enables the compilation of C++03 source code.

Usage

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

Default

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

Related references

7.5 --anachronisms, --no_anachronisms on page 7-272.

7.19 --c90 on page 7-290.

7.20 --c99 on page 7-291.

7.73 --gnu on page 7-350.

7.156 --strict, --no_strict on page 7-439.

7.26 --cpp11 on page 7-298.

7.27 --cpp_compat on page 7-299.

1.2 Source language modes of the compiler on page 1-29.

2.7 Filename suffixes recognized by the compiler on page 2-47.

7.26 --cpp11

Enables the compilation of C++11 source code.

Usage

This option can also be combined with other source language command-line options. For example, `armcc --cpp11 --cpp_compat`.

When compiling C++11 code you must use the `--cpp11` command line option. The default source language mode when a C++ filename suffix is detected is C++03.

Default

For files with a suffix of `.cpp`, `.cxx`, `.c++`, `.cc`, or `.CC`, `--cpp` applies by default.

Related references

7.19 --c90 on page 7-290.
7.20 --c99 on page 7-291.
7.73 --gnu on page 7-350.
7.25 --cpp on page 7-297.
7.156 --strict, --no_strict on page 7-439.
7.27 --cpp_compat on page 7-299.
1.2 Source language modes of the compiler on page 1-29.
2.7 Filename suffixes recognized by the compiler on page 2-47.
10.13 C++11 supported features on page 10-724.

7.27 --cpp_compat

Compiles C++ code to maximize binary compatibility.

Usage

Use `--cpp11 --cpp_compat` to compile C++11 source code using a subset of features that maximizes compatibility with source code compiled with C++03.

Use `--cpp --cpp_compat` to compile C++03 source code maximizing binary compatibility with C++03 code compiled using older compiler versions.

The `--cpp11 --cpp_compat` options behave in the same way as `--cpp11` with the following restrictions:

- When the `--exceptions` option is selected, the array `new` operator with a length not known at compile time does not perform bounds checking. This means that `std::bad_alloc` is thrown if there is an error rather than `std::bad_array_new_length`.
- When the `--exceptions` option is selected, the delegating constructors language feature is disabled. Any use of delegating constructors when the `--cpp_compat` is selected results in an error message.

`armcc` passes the `--cpp_compat` option to `armlink` if it invokes `armlink` to perform a final link step.

You can combine the `--cpp_compat` option with other source language command-line options. For example:

- `armcc --cpp_compat --cpp11`
- `armcc --cpp_compat --cpp`
- `armcc --cpp_compat --cpp --gnu`

Examples

When compiling with `--cpp11 --exceptions`, code can catch `std::bad_array_new_length`:

```
void variable_length_array_new(unsigned i) {
    bool exception_thrown = false;
    try {
        new int[i + 0x40000000];
    } catch (std::bad_array_new_length e)
}
```

When compiling with `--cpp11 --cpp_compat --exceptions`, code can only catch `std::bad_alloc`:

```
void variable_length_array_new_cpp_compat(unsigned i) {
    bool exception_thrown = false;
    try {
        new int[i + 0x40000000];
    } catch (std::bad_alloc e)
}
```

Default

By default, the `--cpp_compat` option is not enabled.

For files with a suffix of `.cpp`, `.cxx`, `.c++`, `.cc`, or `.CC`, the `--cpp` option applies by default.

Related references

7.19 --c90 on page 7-290.
7.20 --c99 on page 7-291.
7.73 --gnu on page 7-350.
7.25 --cpp on page 7-297.
7.156 --strict, --no_strict on page 7-439.
7.26 --cpp11 on page 7-298.
1.2 Source language modes of the compiler on page 1-29.

2.7 Filename suffixes recognized by the compiler on page 2-47.

10.13 C++11 supported features on page 10-724.

7.28 --cpu=list

Lists the architecture and processor names that are supported by the `--cpu=name` option.

Syntax

```
--cpu=list
```

Related references

7.29 --cpu=name compiler option on page 7-302.

4.53 Processors and their implicit Floating-Point Units (FPUs) on page 4-171.

7.29 --cpu=name compiler option

Enables code generation for the selected ARM processor

Syntax

`--cpu=`*`name`*

Where *`name`* is the name of a processor. Enter *`name`* as shown on ARM data sheets, for example, `Cortex-M3`.

Processor names are not case-sensitive.

Default

`armcc` assumes `--cpu=ARM7TDMI` if you do not specify a `--cpu` option.

Usage

The following general points apply to processor options:

Processors

- Selecting the processor selects the appropriate architecture, *Floating-Point Unit* (FPU), and memory organization.
- If you specify a processor for the `--cpu` option, the generated code is optimized for that processor. This enables the compiler to use specific coprocessors or instruction scheduling for optimum performance.

FPU

- Some specifications of `--cpu` imply an `--fpu` selection.
  For example, when building with the `--arm` option, `--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 generated object code incompatible with other ARM processors. For example, code generated for architecture ARMv6 might not run on an ARM920T processor, if the generated object code includes instructions specific to ARMv6. Therefore, you must choose the lowest common denominator processor suited to your purpose.

- If you are building 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 build for Thumb, that is with the `--thumb` option on the command line, the compiler generates 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 generating code for a 16-bit Thumb processor and using VFP, any function containing floating-point operations is compiled for ARM.

Restrictions

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

Related references

7.6 --apcs=qualifier...qualifier on page 7-273.

7.28 --cpu=list on page 7-301.

7.69 --fpu=name compiler option on page 7-344.

7.160 --thumb on page 7-444.

9.14 __smc on page 9-530.

Related information

SMC.

7.30 --create_pch=filename

Instructs the compiler to create a *Precompiled Header* (PCH) file with the specified filename.

Note

This option is deprecated.

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

This option takes precedence over all other PCH options.

Syntax

`--create_pch=`*`filename`*

Where:

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

Related concepts

3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.

3.21 Precompiled Header (PCH) files on page 3-88.

Related references

7.129 --pch on page 7-411.

7.130 --pch_dir=dir on page 7-412.

7.131 --pch_messages, --no_pch_messages on page 7-413.

7.132 --pch_verbose, --no_pch_verbose on page 7-414.

7.166 --use_pch=filename on page 7-451.

9.85 #pragma hdrstop on page 9-605.

9.90 #pragma no_pch on page 9-610.

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

Defines the macro *`name`*.

Syntax

`-D`*`name`*`[(`*`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.

`=`*`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.

Usage

Specifying a macro and a definition with `-D`*`macro`*`=`*`def`* has the same effect as placing the text `#define` *`macro def`* at the head of each source file.

Specifying a macro without a definition with `-D`*`macro`* has the same effect as placing the text `#define` *`macro`* `1` at the head of each source file.

Restrictions

The compiler defines and undefines macros in the following order:

1. Compiler predefined macros.
2. Macros defined explicitly, using `-D`*`name`*.
3. Macros explicitly undefined, using `-U`*`name`*.

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.

Related references

7.18 -C on page 7-289.
7.53 -E on page 7-327.
7.163 -Uname on page 7-447.
9.158 Predefined macros on page 9-697.

7.32 --data_reorder, --no_data_reorder

Enables and 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.

Default

The default is optimization-level dependent:

`-O0`:
: `--no_data_reorder`

`-O1`, `-O2`, `-O3`:
: `--data_reorder`

Related concepts

3.32 Default compiler options that are affected by optimization level on page 3-102.

Related references

7.119 -Onum on page 7-399.

7.33 --debug, --no_debug

Enables and disables the generation of debug tables.

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

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
```

Related references

7.34 --debug_macros, --no_debug_macros on page 7-308.

7.51 --dwarf2 on page 7-325.

7.52 --dwarf3 on page 7-326.

7.119 -Onum on page 7-399.

7.34 --debug_macros, --no_debug_macros

Enables and disables the generation of debug table entries for preprocessor macro definitions.

Usage

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

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

Default

The default is `--debug_macros`.

Related references

7.33 --debug, --no_debug on page 7-307.
7.74 --gnu_defaults on page 7-351.

7.35 --default_extension=ext

Changes the filename extension for object files from the default extension (`.o`) to an extension of your choice.

Syntax

`--default_extension=`*`ext`*

Where:

`ext`
is the filename extension of your choice.

Default

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

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
```

————————————

7.36 --dep_name, --no_dep_name

Enables and 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. ARM does not recommend its use.

Mode

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

Default

The default is `--dep_name`.

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.

Errors

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

Related references

7.128 --parse_templates, --no_parse_templates on page 7-410.
10.9 Template instantiation in ARM C++ on page 10-719.

7.37 --depend=filename

Writes makefile dependency lines to a file during compilation.

Syntax

`--depend=`*`filename`*

Where:

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

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.

Related references

7.39 --depend_format=string on page 7-313.
7.38 --depend_dir=directory_name on page 7-312.
7.41 --depend_system_headers, --no_depend_system_headers on page 7-315.
7.42 --depend_target=target on page 7-316.
7.40 --depend_single_line, --no_depend_single_line on page 7-314.
7.134 --phony_targets on page 7-416.
7.80 --ignore_missing_headers on page 7-357.
7.98 --list on page 7-376.
7.108 -M on page 7-387.
7.109 --md on page 7-388.

7.38 --depend_dir=directory_name

Specifies the directory for dependency output files.

Examples

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

This command outputs the following files:

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

If you specify a dependency file, `--depend=deps`, then the dependency file accumulates the dependency lines from each source file, for example:

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

This command outputs the following files:

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

Related references

7.37 --depend=filename on page 7-311.

7.10 --asm_dir=directory_name on page 7-280.

7.99 --list_dir=directory_name on page 7-378.

7.126 --output_dir=directory_name on page 7-408.

7.39 --depend_format=string

Specifies the format of output dependency files, for compatibility with some UNIX make programs.

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.

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 `\`.

`unix_escaped`
: On Windows systems, `--depend_format=unix_escaped` forces UNIX-style path names, and escapes spaces with `\`.

`unix_quoted`
: On Windows systems, `--depend_format=unix_quoted` forces UNIX-style path names and surrounds them with `""`.

Default

If you do not specify a `--depend_format` option, then the format of output dependency files is either Windows-style paths or UNIX-style paths, whichever is given.

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
```

Related references

7.37 --depend=filename on page 7-311.

7.41 --depend_system_headers, --no_depend_system_headers on page 7-315.

7.42 --depend_target=target on page 7-316.

7.80 --ignore_missing_headers on page 7-357.

7.108 -M on page 7-387.

7.109 --md on page 7-388.

7.134 --phony_targets on page 7-416.

7.40 --depend_single_line, --no_depend_single_line

Specifies 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.

Default

The default is `--no_depend_single_line`.

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
```

Related references

7.41 --depend_system_headers, --no_depend_system_headers on page 7-315.
7.37 --depend=filename on page 7-311.
7.39 --depend_format=string on page 7-313.
7.42 --depend_target=target on page 7-316.
7.80 --ignore_missing_headers on page 7-357.
7.108 -M on page 7-387.
7.109 --md on page 7-388.
7.134 --phony_targets on page 7-416.

7.41 --depend_system_headers, --no_depend_system_headers

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

Default

The default is `--depend_system_headers`.

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
```

Related references

7.40 --depend_single_line, --no_depend_single_line on page 7-314.
7.37 --depend=filename on page 7-311.
7.39 --depend_format=string on page 7-313.
7.42 --depend_target=target on page 7-316.
7.80 --ignore_missing_headers on page 7-357.
7.108 -M on page 7-387.
7.109 --md on page 7-388.
7.134 --phony_targets on page 7-416.

7.42 --depend_target=target

Specifies the target for makefile dependency generation.

Usage

Use this option to override the default target.

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.

Related references

7.37 --depend=filename on page 7-311.
7.39 --depend_format=string on page 7-313.
7.41 --depend_system_headers, --no_depend_system_headers on page 7-315.
7.80 --ignore_missing_headers on page 7-357.
7.108 -M on page 7-387.
7.109 --md on page 7-388.
7.134 --phony_targets on page 7-416.

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

Sets diagnostic messages that have a specific tag to Error severity.

Note

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

Syntax

`--diag_error=`*`tag[,tag,…]`*

Where *`tag`* can be:

- A diagnostic message number to set to error severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `warning`, to treat all warnings as errors.

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 `C`*`nnnn`*`W`.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

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

Sets diagnostic messages that have a specific tag to Remark severity.

The `--diag_remark` option behaves analogously to `--diag_error`, 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` option to display these messages.

Note

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

Syntax

`--diag_remark=`*`tag[,tag,…]`*

Where *`tag`* is a comma-separated list of diagnostic message numbers. This is the four-digit number, *`nnnn`*, with the tool letter prefix, but without the letter suffix indicating the severity.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

7.45 --diag_style=arm|ide|gnu compiler option

Specifies the display style for diagnostic messages.

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`.

Default

The default is `--diag_style=arm`.

Usage

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

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`.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

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

Suppresses diagnostic messages that have a specific tag.

Behaves analogously to `--diag_error`, 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`.

Syntax

`--diag_suppress=`*`tag[,tag,…]`*

Where *`tag`* can be:

- A diagnostic message number to be suppressed. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to suppress all errors that can be downgraded.
- `warning`, to suppress all warnings.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

7.47 --diag_suppress=optimizations

Suppresses diagnostic messages for high-level optimizations.

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.

————————————

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.

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.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

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

Sets diagnostic messages that have a specific tag to Warning severity.

The `--diag_warning` option behaves analogously to `--diag_error`, 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`.

Syntax

`--diag_warning=`*`tag[,tag,…]`*

Where *`tag`* can be:

- A diagnostic message number to set to warning severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to set all errors that can be downgraded to warnings.

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.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

7.49 --diag_warning=optimizations

Sets high-level optimization diagnostic messages to have Warning severity.

Default

By default, optimization messages have Remark severity.

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.

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.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

7.50 --dollar, --no_dollar

Enables and disables the use of dollar signs, `$`, in identifiers.

Default

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

Related references

7.156 --strict, --no_strict on page 7-439.

8.19 Dollar signs in identifiers on page 8-484.

7.157 --strict_warnings on page 7-440.

7.51 --dwarf2

Uses DWARF 2 debug table format.

Default

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

Related references

7.52 --dwarf3 on page 7-326.

Related information

The DWARF Debugging Standard, http://dwarfstd.org/.

7.52 --dwarf3

Uses DWARF 3 debug table format.

Default

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

Related references

7.51 --dwarf2 on page 7-325.

Related information

The DWARF Debugging Standard, http://dwarfstd.org/.

7.53 -E

Executes the preprocessor step only.

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` fails to include such definitions.

Example

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

Related references

7.18 -C on page 7-289.
7.100 --list_macros on page 7-379.
7.109 --md on page 7-388.
7.118 -o filename on page 7-397.
7.121 --old_style_preprocessing on page 7-403.
7.127 -P on page 7-409.

Related information

Why does armcc -E preprocessing result in linker undefined symbol error?.

7.54 --echo

Displays the complete expanded command line, and any separate commands that invoke other external applications, such as `armasm` or `armlink`.

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]
```

7.55 --emit_frame_directives, --no_emit_frame_directives

Places DWARF `FRAME` directives into disassembly output.

Default

The default is `--no_emit_frame_directives`.

Examples

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

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

Related references

7.9 --asm on page 7-279.

7.149 -S on page 7-431.

Related information

Frame directives.

7.56 --enum_is_int

Forces the size of all enumeration types to be at least four bytes.

Note

ARM does not recommend the `--enum_is_int` option for general use.

Default

This option is switched off by default. The smallest data type that can hold the values of all enumerators is used.

Related references

7.88 --interface_enums_are_32_bit on page 7-365.

10.4 Structures, unions, enumerations, and bitfields in ARM C and C++ on page 10-710.

7.57 --errors=filename

Redirects the output of diagnostic messages from stderr to the specified errors file.

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`*.

Usage

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

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

7.58 --exceptions, --no_exceptions

Enables and 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 though C functions.

Default

The default is `--no_exceptions`.

Related references

10.11 C++ exception handling in ARM C++ on page 10-722.

7.59 --exceptions_unwind, --no_exceptions_unwind on page 7-333.

9.83 #pragma exceptions_unwind, #pragma no_exceptions_unwind on page 9-603.

7.59 --exceptions_unwind, --no_exceptions_unwind

Enables and 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.

Default

The default is `--exceptions_unwind`.

Related references

10.11 C++ exception handling in ARM C++ on page 10-722.

7.58 --exceptions, --no_exceptions on page 7-332.

9.83 #pragma exceptions_unwind, #pragma no_exceptions_unwind on page 9-603.

7.60 --execute_only

Generates execute-only code by adding the `EXECONLY` attribute to the `AREA` directive for all code sections, preventing the compiler from generating any data accesses to code sections.

To keep code and data in separate sections, the compiler disables literal pools and branch tables. That is, specifying `--execute_only` implicitly specifies the following compiler options:

- `--no_integer_literal_pools`.
- `--no_float_literal_pools`.
- `--no_string_literal_pools`.
- `--no_branch_tables`.

Restrictions

Execute-only code must be Thumb code.

Execute-only code is only supported for:

- Processors that support the ARMv7-M architecture, such as Cortex-M3, Cortex-M4, and Cortex-M7.
- Processors that support the ARMv6-M architecture.

  Note

  ARM has only performed limited testing of execute-only code on ARMv6-M targets.

If your application calls library functions, the library objects included in the image are not execute-only compliant. You must ensure these objects are not assigned to an execute-only memory region.

Note

ARM does not provide libraries that are built without literal pools. The libraries still use literal pools, even when you use the various `--no_*_literal_pools` options.

Related concepts

3.19 Compiler support for literal pools on page 3-86.

Related references

7.87 --integer_literal_pools, --no_integer_literal_pools on page 7-364.
7.158 --string_literal_pools, --no_string_literal_pools on page 7-441.
7.14 --branch_tables, --no_branch_tables on page 7-284.
7.63 --float_literal_pools, --no_float_literal_pools on page 7-337.

Related information

AREA (assembler directive).
Building applications for execute-only memory.

7.61 --extended_initializers, --no_extended_initializers

Enables and disables 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.

Default

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

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

Related references

7.156 --strict, --no_strict on page 7-439.

7.157 --strict_warnings on page 7-440.

8.16 Constant expressions on page 8-481.

7.62 --feedback=filename

Enables the linker to communicate with the compiler to eliminate unused functions.

Syntax

`--feedback=`*`filename`*

Where:

`filename`

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

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.

Related references

7.155 --split_sections on page 7-438.

2.14 Linker feedback during compilation on page 2-55.

Related information

--feedback_type=type linker option.

7.63 --float_literal_pools, --no_float_literal_pools

Controls whether the compiler places floating-point and vector constants in literal pools.

With the `--float_literal_pools` option, where there are floating-point or vector constants in source code and hardware support is available, the compiler generates code that loads those constants from literal pools using VFP instructions:

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 12 bytes (alignment 4)
    Address: 0x00000000

    $a
    .text
    main
        0x00000000:    ed9f0a00    ....    VLDR     s0,[pc,#0] ; [0x8] = 0x42280000
        0x00000004:    eaffffe    ....    B        f
    $d
        0x00000008:    42280000    ..(B    DCD    1109917696
```

With the `--no_float_literal_pools` option, the compiler generates code that loads these constants using core instruction set loads and reinterprets them as floats or vectors:

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 16 bytes (alignment 4)
    Address: 0x00000000

    $a
    .text
    main
        0x00000000:    e59f0004    ....    LDR      r0,[pc,#4] ; [0xc] = 0x42280000
        0x00000004:    ee000a10    ....    VMOV     s0,r0
        0x00000008:    eaffffe    ....    B        f
    $d
        0x0000000c:    42280000    ..(B    DCD    1109917696
```

If you also specify the `--no_integer_literal_pools` option, the compiler constructs these constants with sequences of `MOVW` and `MOVT` instructions.

This option also controls integer vectors.

Default

The default is `--float_literal_pools`.

`--execute_only` implies `--no_float_literal_pools`, unless `--float_literal_pools` is explicitly specified.

Note

Do not use `--execute_only` in conjunction with `--float_literal_pools`. If you do, then the compiler places the literal pool in an unreadable, execute-only code region.

Related concepts

3.19 Compiler support for literal pools on page 3-86.

Related references

7.87 --integer_literal_pools, --no_integer_literal_pools on page 7-364.
7.158 --string_literal_pools, --no_string_literal_pools on page 7-441.
7.14 --branch_tables, --no_branch_tables on page 7-284.
7.60 --execute_only on page 7-334.

7.64 --force_new_nothrow, --no_force_new_nothrow

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. ARM does not recommend its use.

Mode

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

Default

The default is `--no_force_new_nothrow`.

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();
```

Related references

10.5 Using the ::operator new function in ARM C++ on page 10-715.

7.65 --forceinline

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 never inlined into itself.

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

Related references

7.86 --inline, --no_inline on page 7-363.

9.8 __inline on page 9-523.

9.6 __forceinline on page 9-520.

9.30 __attribute__((always_inline)) function attribute on page 9-549.

7.66 --fp16_format=format

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.

Syntax

`--fp16_format=`*`format`*

Where *`format`* is one of:

`alternative`
: An alternative to `ieee` 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 `__fp16` data type.

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`.

Related concepts

4.47 Compiler and library support for half-precision floating-point numbers on page 4-163.

Related references

7.67 --fpmode=model on page 7-341.

7.67 --fpmode=model

Specifies floating-point standard conformance. This controls which floating-point optimizations the compiler can perform, and also influences library selection.

Syntax

`--fpmode=`*`model`*

Where *`model`* 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 NaNs in arithmetic operations with `--fpmode=std` causes 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.

Default

The default is `--fpmode=std`.

Related concepts

4.45 Limitations on hardware handling of floating-point arithmetic on page 4-160.

Related references

7.69 --fpu=name compiler option on page 7-344.

Related information

ARM Application Note 133 - Using VFP with RVDS.

7.68 --fpu=list

Lists the FPU architectures that are supported by the --fpu=name option.

Deprecated options are not listed.

Related references

7.69 --fpu=name compiler option on page 7-344.

7.69 --fpu=name compiler option

Specifies 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.

The compiler sets a build attribute corresponding to name in the object file. The linker determines compatibility between object files, and selection of libraries, accordingly.

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

Syntax

`--fpu=`*`name`*

Where *`name`* is one of:

`none`
: Selects no floating-point option. No floating-point code is to be used. This makes your object file compatible with other object files built with any FPU.

  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.

`fpv5_d16`
: Selects a hardware floating-point unit conforming to the FPv5-D16 architecture.

`fpv5-sp`
: Selects a hardware floating-point unit conforming to the single precision variant of the FPv5 architecture.

`softvfp`
: Selects software floating-point support where floating-point operations are performed by a floating-point library, `fplib`. This is the default if you do not specify a `--fpu` option, or if you select a processor 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:

- Building with `--thumb` behaves in a similar way to `--fpu=softvfp` except that it links with floating-point libraries that use VFP instructions.
- Building 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.

`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.

`softvfp+fpv5_d16`
: Selects a hardware floating-point unit conforming to FPv5-D16, with software floating-point linkage.

`softvfp+fpv5-sp`
: Selects a hardware floating-point unit conforming to FPv5-SP, with software floating-point linkage.

Usage

Any FPU explicitly selected using the `--fpu` option always overrides any FPU implicitly selected using the `--cpu` option.

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

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 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.

Default

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

If the processor specified with `--cpu` has a VFP coprocessor, the default target FPU architecture is the VFP architecture for that processor. If a VFP coprocessor is present, VFP instructions are generated.

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

Related concepts

4.44 Vector Floating-Point (VFP) architectures on page 4-159.
4.49 Compiler support for floating-point computations and linkage on page 4-165.

Related references

7.6 --apcs=qualifier...qualifier on page 7-273.
7.7 --arm on page 7-277.
7.29 --cpu=name compiler option on page 7-302.
7.67 --fpmode=model on page 7-341.
7.160 --thumb on page 7-444.
9.15 __softfp on page 9-531.

Related information

MRC and MRC2.

7.70 --friend_injection, --no_friend_injection

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. ARM does not recommend its use.

Mode

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

Default

The default is `--no_friend_injection`.

Related references

8.27 friend on page 8-492.

7.71 -g

Enables the generation of debug tables.

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.

Default

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

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

Related references

7.33 --debug, --no_debug on page 7-307.
7.34 --debug_macros, --no_debug_macros on page 7-308.
7.51 --dwarf2 on page 7-325.
7.52 --dwarf3 on page 7-326.
7.119 -Onum on page 7-399.

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

Treats the specified register names as fixed registers, and prevents the compiler from generating code that uses these registers.

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.

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`.

Restrictions

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

Example

```
--global_reg=1,4,5
```

Reserves registers `r4`, `r7` and `r8`

Related references

9.7 __global_reg on page 9-521.

7.73 --gnu

Enables the GNU compiler extensions that the ARM compiler supports.

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.

Usage

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

Related references

7.19 --c90 on page 7-290.
7.20 --c99 on page 7-291.
7.25 --cpp on page 7-297.
7.156 --strict, --no_strict on page 7-439.
7.26 --cpp11 on page 7-298.
7.27 --cpp_compat on page 7-299.
1.2 Source language modes of the compiler on page 1-29.
2.7 Filename suffixes recognized by the compiler on page 2-47.
7.74 --gnu_defaults on page 7-351.
7.76 --gnu_version=version on page 7-353.
8.45 GNU extensions to the C and C++ languages on page 8-510.
9.158 Predefined macros on page 9-697.

7.74 --gnu_defaults

Alters the default settings of certain other options to match the default behavior found in GCC.

Usage

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

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

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

Related references

7.3 --allow_null_this, --no_allow_null_this on page 7-270.
7.34 --debug_macros, --no_debug_macros on page 7-308.
7.73 --gnu on page 7-350.
7.81 --implicit_include, --no_implicit_include on page 7-358.
7.152 --signed_bitfields, --unsigned_bitfields on page 7-435.
7.174 --wchar, --no_wchar on page 7-459.

7.75 --gnu_instrument, --no_gnu_instrument

Inserts GCC-style instrumentation calls for profiling entry and exit to functions.

Note

The `--gnu_instrument` option is deprecated from ARM Compiler 5.05 onwards.

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);
```

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))`.

Related references

9.39 __attribute__((no_instrument_function)) function attribute on page 9-558.

7.76 --gnu_version=version

Attempts to make the compiler compatible with a particular version of GCC.

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.

Mode

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

Usage

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

The maximum supported values for `--gnu_version` in `armcc` are as follows:

armcc	GCC
4.0, 4.1	`40300` (GCC 4.3)
5.0, 5.01, 5.02, 5.03, 5.04	`40400` (GCC 4.4)
5.05, 5.06	`40800` (GCC 4.8)

Default

In ARM Compiler 5.06, the default is `40700`. This corresponds to GCC version 4.7.0.

In ARM Compiler 4.1 through to 5.05, the default is `40200`. This corresponds to GCC version 4.2.0.

Example

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

Related references

7.73 --gnu on page 7-350.

7.77 --guiding_decls, --no_guiding_decls

Enables and 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. ARM does not recommend its use.

Mode

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

Default

The default is `--no_guiding_decls`.

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.

Related references

7.120 --old_specializations, --no_old_specializations on page 7-402.

7.6 --apcs=qualifier...qualifier on page 7-273.

7.78 --help

Displays a summary of the main command-line options.

Default

This is the default if you specify `armcc` without any options or source files.

Related references

7.151 --show_cmdline on page 7-434.

7.172 --vsn on page 7-457.

7.79 -Idir[,dir,...]

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.

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.

Related concepts

2.9 Factors influencing how the compiler searches for header files on page 2-50.

Related references

7.90 -Jdir[,dir,...] on page 7-367.
7.91 --kandr_include on page 7-368.
7.136 --preinclude=filename on page 7-418.
7.159 --sys_include on page 7-443.
2.10 Compiler command-line options and search paths on page 2-51.

7.80 --ignore_missing_headers

Prints dependency lines for header files even if the header files are missing.

This option 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.

Usage

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

Related references

7.37 --depend=filename on page 7-311.
7.39 --depend_format=string on page 7-313.
7.41 --depend_system_headers, --no_depend_system_headers on page 7-315.
7.42 --depend_target=target on page 7-316.
7.108 -M on page 7-387.
7.109 --md on page 7-388.
7.134 --phony_targets on page 7-416.

7.81 --implicit_include, --no_implicit_include

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

Mode

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

Default

The default is `--implicit_include`.

Related references

7.82 --implicit_include_searches, --no_implicit_include_searches on page 7-359.

7.74 --gnu_defaults on page 7-351.

10.9 Template instantiation in ARM C++ on page 10-719.

7.82 --implicit_include_searches, --no_implicit_include_searches

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 `ARMCC5INC` environment variable, and the `ARMINC` environment variable. The search path also includes the default `../include` directory if `-J`, `ARMCC5INC`, 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.

Mode

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

Default

The default is `--no_implicit_include_searches`.

Related references

7.79 -Idir[,dir,...] on page 7-356.
7.81 --implicit_include, --no_implicit_include on page 7-358.
7.90 -Jdir[,dir,...] on page 7-367.
10.9 Template instantiation in ARM C++ on page 10-719.
2.10 Compiler command-line options and search paths on page 2-51.

Related information

Toolchain environment variables.

7.83 --implicit_key_function, --no_implicit_key_function

Controls 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`.

Default

The default is `--implicit_key_function`.

Related references

7.48 --diag_warning=tag[,tag,...] on page 7-322.

7.143 --remarks on page 7-425.

7.84 --implicit_typename, --no_implicit_typename

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. ARM does not recommend its use.

————————————

Mode

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

Default

The default is `--no_implicit_typename`.

———— **Note** ————

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

————————————

Related references

7.36 --dep_name, --no_dep_name on page 7-310.

7.128 --parse_templates, --no_parse_templates on page 7-410.

10.9 Template instantiation in ARM C++ on page 10-719.

7.85 --info=totals

Reports total sizes of the object code and data 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 assembly code sizes when embedded assembly exists in the source code.

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.

Related concepts

4.9 Code metrics on page 4-119.

Related references

7.98 --list on page 7-376.

Related information

--info=topic[,topic,...] fromelf option.

--text fromelf option.

--info=topic[,topic,...] linker option.

7.86 --inline, --no_inline

Enables and disables 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, as the compiler uses a complex decision tree to decide whether to inline a particular function.

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

Default

The default is `--inline`.

Related references

7.11 --autoinline, --no_autoinline on page 7-281.
7.119 -Onum on page 7-399.
7.124 -Ospace on page 7-406.
7.125 -Otime on page 7-407.
9.6 __forceinline on page 9-520.
9.8 __inline on page 9-523.
2.14 Linker feedback during compilation on page 2-55.
7.65 --forceinline on page 7-339.
9.8 __inline on page 9-523.
9.6 __forceinline on page 9-520.
9.30 __attribute__((always_inline)) function attribute on page 9-549.

7.87 --integer_literal_pools, --no_integer_literal_pools

Controls whether the compiler places integer and address constants in literal pools.

With the `--integer_literal_pools` option, when the compiler cannot construct integer and address constants in a single instruction, it often places them in literal pools:

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 12 bytes (alignment 4)
    Address: 0x00000000

    $a
    .text
    f
        0x00000000:    e59f0000    ....    LDR      r0,[pc,#0] ; [0x8] = 0xdeadbeef
        0x00000004:    e12fff1e    ../.    BX       lr
    $d
        0x00000008:    deadbeef    ....    DCD    3735928559
```

The `--no_integer_literal_pools` option instructs the compiler to use sequences of `MOVW` and `MOVT` instructions to construct these constants:

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 12 bytes (alignment 4)
    Address: 0x00000000

    $a
    .text
    f
        0x00000000:    e30b0eef    ....    MOV      r0,#0xbeef
        0x00000004:    e34d0ead    ..M.    MOVT     r0,#0xdead
        0x00000008:    e12fff1e    ../.    BX       lr
```

64-bit integers are constructed with two `MOVW` instructions and two `MOVT` instructions.

Note

You cannot use the `--no_integer_literal_pools` option with target architectures earlier than v6T2.

Default

The default is `--integer_literal_pools`.

`--execute_only` implies `--no_integer_literal_pools`, unless `--integer_literal_pools` is explicitly specified.

Note

Do not use `--execute_only` in conjunction with `--integer_literal_pools`. If you do, then the compiler places the literal pool in an unreadable, execute-only code region.

Related concepts

3.19 Compiler support for literal pools on page 3-86.

Related references

7.158 --string_literal_pools, --no_string_literal_pools on page 7-441.
7.14 --branch_tables, --no_branch_tables on page 7-284.
7.63 --float_literal_pools, --no_float_literal_pools on page 7-337.
7.60 --execute_only on page 7-334.

7.88 --interface_enums_are_32_bit

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

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.)

Default

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

Related references

7.56 --enum_is_int on page 7-330.

7.89 --interleave

Interleaves C or C++ source code line by line as comments within the assembly listing.

Usage

Use the `--interleave` option with the `--asm` option or `-S` option.

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

Table 7-3 Compiling with the --interleave option

Compiler option	Action
`--asm` `--interleave`	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 `-c` 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 `.txt`
`-S --interleave`	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 `.txt`

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`.

Related references

7.9 --asm on page 7-279.

7.149 -S on page 7-431.

7.90 -Jdir[,dir,...]

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 `ARMCC5INC` 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
```

Syntax

`-J`*`dir`*`[,`*`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.

Related concepts

2.9 Factors influencing how the compiler searches for header files on page 2-50.

Related references

7.79 -Idir[,dir,...] on page 7-356.

7.91 --kandr_include on page 7-368.

7.136 --preinclude=filename on page 7-418.

7.159 --sys_include on page 7-443.

2.10 Compiler command-line options and search paths on page 2-51.

Related information

Toolchain environment variables.

7.91 --kandr_include

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.

Default

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

Related concepts

2.9 Factors influencing how the compiler searches for header files on page 2-50.

Related references

7.90 -Jdir[,dir,...] on page 7-367.
7.79 -Idir[,dir,...] on page 7-356.
7.136 --preinclude=filename on page 7-418.
7.159 --sys_include on page 7-443.
2.10 Compiler command-line options and search paths on page 2-51.

7.92 -Lopt

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.

Syntax

`-L`*`opt`*

Where:

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

Restrictions

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

Example

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

Related references

7.1 -Aopt on page 7-268.

7.151 --show_cmdline on page 7-434.

7.93 --library_interface=lib

Generates code that is compatible with the selected library type.

Syntax

`--library_interface=`*`lib`*

Where *`lib`* is one of:

`none`

Specifies that the compiler output works with any ISO C90 library.

In general, the compiler avoids the use of AEABI-defined library functions. For example, this option suppresses the use of AEABI-defined functions that are introduced only as an optimization such as `__aeabi_memcpy`.

AEABI-defined library functions are only used to handle operations that do not have a short machine code equivalent. For example, the `__aeabi_uidiv` function is used for integer division where there is no divide instruction available in the target instruction set.

`armcc`

Specifies that the compiler output works with the ARM runtime libraries in ARM Compiler 4.1 and later.

`armcc_c90`

Behaves similarly to `--library_interface=armcc`. 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 `math.h` function names might be prefixed with `__hardfp_`, for example `__hardfp_tgamma`.

`aeabi_clib90`

Specifies that the compiler output works with any ISO C90 library compliant with the *ARM Embedded Application Binary Interface* (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 `--library_interface=aeabi_clib` is equivalent to specifying either `--library_interface=aeabi_clib90` or `--library_interface=aeabi_clib99`, 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.

`rvct30`

Specifies that the compiler output is compatible with RVCT 3.0 runtime libraries.

`rvct30_c90`

Behaves similarly to `rvct30`. In addition, specifies that the compiler output is compatible with any ISO C90 library.

`rvct31`

Specifies that the compiler output is compatible with RVCT 3.1 runtime libraries.

`rvct31_c90`

Behaves similarly to `rvct31`. In addition, specifies that the compiler output is compatible with any ISO C90 library.

`rvct40`

Specifies that the compiler output is compatible with RVCT 4.0 runtime libraries.

`rvct40_c90`
Behaves similarly to `rvct40`. In addition, specifies that the compiler output is compatible with any ISO C90 library.

Default

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

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.

Note

`_hardfp` options are not supported by ARM C libraries.

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.

Related information

Compliance with the Application Binary Interface (ABI) for the ARM architecture.

7.94 --library_type=lib

Enables the selected library to be used at link time.

Note

This option can be overridden at link time by providing it to the linker.

Syntax

`--library_type=`*`lib`*

Where *`lib`* is one of:

`standardlib`

Specifies that the full ARM 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.

Default

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

Related information

About microlib.
Building an application with microlib.
--library_type=lib linker option.

7.95 --liclinger=seconds

The time in seconds that a license is to remain checked out.

Syntax

`--liclinger=`*`seconds`*

7.96 --licretry

If you are using floating licenses, `armcc` makes up to 10 attempts to obtain a license when invoked.

Note

This option is always enabled. `armcc` ignores this option if you specify it.

Related information

ARM DS-5 License Management Guide.

--licretry assembler option.

--licretry fromelf option.

Toolchain environment variables.

--licretry linker option.

7.97 --link_all_input, --no_link_all_input

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.

Default

The default is `--no_link_all_input`.

Related references

7.23 --compile_all_input, --no_compile_all_input on page 7-295.

2.7 Filename suffixes recognized by the compiler on page 2-47.

7.98 --list

Generates 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.

Usage

Typically, you use raw listing information 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.

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}
```

Related references

7.9 --asm on page 7-279.
7.17 -c on page 7-288.
7.37 --depend=filename on page 7-311.
7.39 --depend_format=string on page 7-313.
7.85 --info=totals on page 7-362.
7.89 --interleave on page 7-366.
7.99 --list_dir=directory_name on page 7-378.
7.109 --md on page 7-388.
7.149 -S on page 7-431.
5.1 Severity of compiler diagnostic messages on page 5-206.

7.99 --list_dir=directory_name

Specifies a directory for `--list` output.

Example

```
armcc -c --list_dir=lst --list f1.c f2.c
```

Result:

```
lst/f1.lst
lst/f2.lst
```

Related references

7.10 --asm_dir=directory_name on page 7-280.

7.38 --depend_dir=directory_name on page 7-312.

7.98 --list on page 7-376.

7.126 --output_dir=directory_name on page 7-408.

7.100 --list_macros

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.

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.

Restrictions

Code generation is suppressed.

Related references

9.158 Predefined macros on page 9-697.

7.31 -Dname[(parm-list)][=def] on page 7-305.

7.53 -E on page 7-327.

7.151 --show_cmdline on page 7-434.

7.170 --via=filename on page 7-455.

7.101 --littleend

Generates code suitable for an ARM processor using little-endian memory.

With little-endian memory, the least significant byte of a word has the lowest address.

Default

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

Related references

7.12 --bigend on page 7-282.

7.102 --locale=lang_country

Specifies the locale for source files.

Syntax

`--locale=`*`lang_country`*

Where:

`lang_country`
is the new default locale.

Use this option in combination with `--multibyte_chars`.

Default

If you do not specify this option, the system locale is used.

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.

Note

If the source file encoding is UTF-8 or UTF-16, and the file starts with a byte order mark then the compiler ignores the `--locale` and `--[no_]multibyte_chars` options and interprets the file as UTF-8 or UTF-16.

Example

To compile Japanese source files on an English-based Windows workstation, use:

```
--multibyte_chars --locale=japanese
```

Related references

7.110 --message_locale=lang_country[.codepage] on page 7-389.
7.113 --multibyte_chars, --no_multibyte_chars on page 7-392.

7.103 --long_long

Permits use of the **`long long`** data type in strict mode.

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;
}
```

Related references

7.156 --strict, --no_strict on page 7-439.

7.104 --loop_optimization_level=opt

Trades code size for performance by controlling how much loop optimization the compiler performs.

The compiler can use several different techniques for specifically targeting loop optimizations, such as loop unrolling and inlining. However, these techniques can impact code size.

Syntax

`--loop_optimization_level=`*`opt`*

Where *`opt`* is one of:

`0`
: Specifies that the compiler does not perform any loop optimization. This option is usually best for code size.

`1`
: Specifies that the compiler performs some loop optimization. This option tries to balance code size and performance.

`2`
: Specifies that the compiler performs high-level optimization, including aggressive loop optimization. This option is usually best for performance.

Restrictions

This option can only be used when both `-O3` and `-Otime` options are given. That is:

```
armcc -O3 -Otime --loop_optimization_level=2 ...
```

Default

The default is 1.

Specifying `-O3 -Otime` implies `--loop_optimization_level=1`.

Related concepts

4.20 Inline functions on page 4-131.

4.7 Loop unrolling in C code on page 4-115.

Related references

7.119 -Onum on page 7-399.

7.125 -Otime on page 7-407.

7.105 --loose_implicit_cast

Makes illegal implicit casts legal, such as implicit casts of a nonzero integer to a pointer.

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.

7.106 --lower_ropi, --no_lower_ropi

Enables and disables less restrictive C when compiling with `--apcs=/ropi`.

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.

Related concepts

2.13 Code compatibility between separately compiled and assembled modules on page 2-54.

Related references

7.107 --lower_rwpi, --no_lower_rwpi on page 7-386.

7.6 --apcs=qualifier...qualifier on page 7-273.

7.107 --lower_rwpi, --no_lower_rwpi

Enables and disables less restrictive C and C++ when compiling with `--apcs=/rwpi`.

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.

Related concepts

2.13 Code compatibility between separately compiled and assembled modules on page 2-54.

Related references

7.106 --lower_ropi, --no_lower_ropi on page 7-385.

7.6 --apcs=qualifier...qualifier on page 7-273.

7.108 -M

Produces 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.

Example

You can redirect output to a file by using standard UNIX and MS-DOS notation, for example:

```
armcc -M source.c > Makefile
```

Related references

7.18 -C on page 7-289.
7.37 --depend=filename on page 7-311.
7.41 --depend_system_headers, --no_depend_system_headers on page 7-315.
7.53 -E on page 7-327.
7.109 --md on page 7-388.
7.40 --depend_single_line, --no_depend_single_line on page 7-314.
7.118 -o filename on page 7-397.

7.109 --md

Creates makefile dependency lists.

Make utilities use makefile dependency lists to determine dependencies between files, for example to determine header file dependencies.

The compiler names the makefile dependency list `*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).

Related references

7.37 --depend=filename on page 7-311.
7.39 --depend_format=string on page 7-313.
7.41 --depend_system_headers, --no_depend_system_headers on page 7-315.
7.53 -E on page 7-327.
7.108 -M on page 7-387.
7.40 --depend_single_line, --no_depend_single_line on page 7-314.
7.118 -o filename on page 7-397.

7.110 --message_locale=lang_country[.codepage]

Specifies the language for error and warning messages.

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`.
- `ja_JP`.

Default

If you do not specify `--message_locale`, the compiler assumes `--message_locale=en_US`.

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.

Errors

If you specify a setting that is not supported, the compiler generates an error message.

Example

To display messages in Japanese, use:

```
--message_locale=ja_JP
```

Related references

7.102 --locale=lang_country on page 7-381.
7.113 --multibyte_chars, --no_multibyte_chars on page 7-392.

7.111 --min_array_alignment=opt

Specifies the minimum alignment of arrays.

Syntax

`--min_array_alignment=`*`opt`*

Where:

`opt`

specifies the minimum alignment of arrays. The value of *`opt`* is one of:

`1`

byte alignment, or unaligned

`2`

two-byte, halfword alignment

`4`

four-byte, word alignment

`8`

eight-byte, doubleword alignment.

Usage

ARM does not recommend using this option, 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.

Default

If you do not use this option, arrays are unaligned (byte aligned).

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
```

Related references

9.2 __align on page 9-516.

9.3 __ALIGNOF__ on page 9-517.

7.112 --mm

This option has the same effect as `-M --no_depend_system_headers`.

Related references

7.108 -M on page 7-387.

7.41 --depend_system_headers, --no_depend_system_headers on page 7-315.

7.113 --multibyte_chars, --no_multibyte_chars

Enables and disables processing for multibyte character sequences in comments, string literals, and character constants.

Default

`--multibyte_chars` is the default, but the option only has an effect in locales that use multibyte characters.

Usage

Multibyte encodings are used for character sets such as the Japanese *Shift-Japanese Industrial Standard* (Shift-JIS).

Note

If the source file encoding is UTF-8 or UTF-16, and the file starts with a byte order mark then the compiler ignores the `--locale` and `--[no_]multibyte_chars` options and interprets the file as UTF-8 or UTF-16.

Related references

7.102 --locale=lang_country on page 7-381.

7.110 --message_locale=lang_country[.codepage] on page 7-389.

7.114 --multifile, --no_multifile

Enables and 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.

To meet the requirements of standard make systems, an empty object file is created for each subsequent source file specified on the command line. However, only a single combined object file is created if you also specify `-o filename`.

Note

Compiling with `--multifile` has no effect if only a single source file is specified on the command line.

Default

The default is `--no_multifile`.

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. However, depending on the number of source files and structure of the program, the compiler might require significantly more memory and significantly more compilation time. For the best optimization results, choose small groups of functionally related source files.

As a guideline, you can expect `--multifile` to scale well up to modules in the low hundreds of thousands of lines of code.

Example

```
armcc -c --multifile test1.c ... testn.c -o test.o
```

Because `-o` is used, a single combined object file named `test.o` is created..

Related references

7.17 -c on page 7-288.
7.35 --default_extension=ext on page 7-309.
7.118 -o filename on page 7-397.
7.119 -Onum on page 7-399.
7.177 --whole_program on page 7-462.
9.158 Predefined macros on page 9-697.

7.115 --multiply_latency=cycles

Tells the compiler the number of cycles used by the hardware multiplier.

Syntax

`--multiply_latency=`*`cycles`*

Where *`cycles`* is the number of cycles used.

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 lets you convey the correct number of cycles for a given processor.

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.

Example

```
--multiply_latency=33
```

Related information

MUL.

7.116 --narrow_volatile_bitfields

Accesses volatile bitfields using the smallest access size that contains the entire bitfield.

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.

Related information

Application Binary Interface (ABI) for the ARM Architecture.

7.117 --nonstd_qualifier_deduction, --no_nonstd_qualifier_deduction

Controls whether or not nonstandard template argument deduction is 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. ARM does not recommend its use.

Mode

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

Default

The default is `--no_nonstd_qualifier_deduction`.

7.118 -o filename

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 the following tables.

Syntax

If you specify a `-o` option, the compiler names the output file according to the conventions described by the following table.

Table 7-4 Compiling with the -o option

Compiler option	Action	Usage notes
`-o-`	writes output to the standard output stream	*filename* is `-`.`-S` is assumed unless `-E` is specified.
`-o` *filename*	produces an executable image with name *filename*	
`-c -o` *filename*	produces an object file with name *filename*	
`-S -o` *filename*	produces an assembly language file with name *filename*	
`-E -o` *filename*	produces a file containing preprocessor output with name *filename*	

Note

This option overrides the `--default_extension` option.

Default

If you do not specify a `-o` option, the compiler names the output file according to the conventions described by the following table.

Table 7-5 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

Related references

7.9 --asm on page 7-279.
7.17 -c on page 7-288.
7.35 --default_extension=ext on page 7-309.
7.37 --depend=filename on page 7-311.
7.39 --depend_format=string on page 7-313.
7.53 -E on page 7-327.
7.89 --interleave on page 7-366.
7.98 --list on page 7-376.

7.109 --md on page 7-388.

7.149 -S on page 7-431.

7.119 -Onum

Specifies the level of optimization to be used when compiling source files.

Syntax

`-Onum`

Where *num* is one of the following:

0

Minimum optimization. Turns off most optimizations. When debugging is enabled, this option gives the best possible debug view because the structure of the generated code directly corresponds to the source code. All optimization that interferes with the debug view is disabled. In particular:
- Breakpoints can be set on any reachable point, including dead code.
- The value of a variable is available everywhere within its scope, except where it is uninitialized.
- Backtrace gives the stack of open function activations that is expected from reading the source.

Note

Although the debug view produced by `-O0` corresponds most closely to the source code, users might prefer the debug view produced by `-O1` because this improves the quality of the code without changing the fundamental structure.

Note

Dead code includes reachable code that has no effect on the result of the program, for example an assignment to a local variable that is never used. Unreachable code is specifically code that cannot be reached via any control flow path, for example code that immediately follows a return statement.

1

Restricted optimization. The compiler only performs optimizations that can be described by debug information. Removes unused inline functions and unused static functions. Turns off optimizations that seriously degrade the debug view. If used with `--debug`, this option gives a generally satisfactory debug view with good code density.
The differences in the debug view from `–O0` are:
- Breakpoints cannot be set on dead code.
- Values of variables might not be available within their scope after they have been initialized. For example if their assigned location has been reused.
- Functions with no side-effects might be called out of sequence, or might be omitted if the result is not needed.
- Backtrace might not give the stack of open function activations that is expected from reading the source because of the presence of tailcalls.

The optimization level `–O1` produces good correspondence between source code and object code, especially when the source code contains no dead code. The generated code can be significantly smaller than the code at `–O0`, which can simplify analysis of the object code.

`2`

High optimization. If used with `--debug`, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler might perform optimizations that cannot be described by debug information.

This is the default optimization level.

The differences in the debug view from `–O1` are:

- The source code to object code mapping might be many to one, because of the possibility of multiple source code locations mapping to one point of the file, and more aggressive instruction scheduling.
- Instruction scheduling is allowed to cross sequence points. This can lead to mismatches between the reported value of a variable at a particular point, and the value you might expect from reading the source code.
- The compiler automatically inlines functions.

`3`

Maximum optimization. When debugging is enabled, this option typically gives a poor debug view. ARM recommends debugging at lower optimization levels.
If you use `-O3` and `-Otime` together, the compiler performs extra optimizations that are more aggressive, such as:

- High-level scalar optimizations, including loop unrolling. 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.

These optimizations effectively rewrite the input source code, resulting in object code with the lowest correspondence to source code and the worst debug view. The `--loop_optimization_level=option` controls the amount of loop optimization performed at `–O3 –Otime`. The higher the amount of loop optimization the worse the correspondence between source and object code.

Use of the `--vectorize` option also lowers the correspondence between source and object code.

For extra information about the high level transformations performed on the source code at `–O3 –Otime` use the `--remarks` command-line option.

—————— **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.

————————————

—————— **Note** ——————

By default, the compiler optimizes to reduce image size at the expense of a possible increase in execution time. That is, `-Ospace` is the default, rather than `-Otime`. Note that `-Ospace` is not affected by the optimization level `-Onum`. That is, `-O3 -Ospace` enables more optimizations than `-O2 -Ospace`, but does not perform more aggressive size reduction.

————————————

Default

If you do not specify `-O`*`num`*, the compiler assumes `-O2`.

Related concepts

4.1 The compiler as an optimizing compiler on page 4-106.
4.3 Compiler optimization levels and the debug view on page 4-108.

Related references

7.11 --autoinline, --no_autoinline on page 7-281.
7.33 --debug, --no_debug on page 7-307.
7.65 --forceinline on page 7-339.
7.67 --fpmode=model on page 7-341.
7.86 --inline, --no_inline on page 7-363.
7.114 --multifile, --no_multifile on page 7-393.
7.124 -Ospace on page 7-406.
7.125 -Otime on page 7-407.
7.104 --loop_optimization_level=opt on page 7-383.

7.120 --old_specializations, --no_old_specializations

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. ARM does not recommend its use.

Mode

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

Default

The default is `--no_old_specializations`.

7.121 --old_style_preprocessing

Performs preprocessing in the style of legacy compilers that do not follow the ISO C Standard.

Related references

7.53 -E on page 7-327.

7.122 --omf_browse

Enables the generation and storing of source browser information.

Syntax

`--omf_browse=`*`filename`*`.crf`

Where:

`filename`
is the name of the file where the source browser information is stored.

7.123 --ool_section_name, --no_ool_section_name

Controls whether `#pragma arm section code` affects out-of-line copies of inline functions.

The `#pragma arm section code` pragma places functions in a separate named section.

With `--no_ool_section_name`, the compiler ignores this pragma for inline functions. Out-of-line copies of inline functions are placed in the `.text` section.

With `--ool_section_name`, the compiler respects the pragma for inline functions. Out-of-line copies of inline functions are placed in the specified section.

Default

The default is `--ool_section_name`.

Related references

9.77 #pragma arm section [section_type_list] on page 9-596.

7.124 -Ospace

Performs 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`.

Default

If you do not specify either `-Ospace` or `-Otime`, the compiler assumes `-Ospace`.

Related references

9.91 #pragma Onum on page 9-611.
9.94 #pragma Otime on page 9-614.
9.93 #pragma Ospace on page 9-613.
7.119 -Onum on page 7-399.
7.125 -Otime on page 7-407.

7.125 -Otime

Performs 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`.

Default

If you do not specify `-Otime`, the compiler assumes `-Ospace`.

Example

When the `-Otime` option is selected, the compiler compiles:

```
while (expression) body;
```

as:

```
if (expression)
{
    do body;
    while (expression);
}
```

Related references

7.114 --multifile, --no_multifile on page 7-393.

7.119 -Onum on page 7-399.

7.124 -Ospace on page 7-406.

9.91 #pragma Onum on page 9-611.

9.93 #pragma Ospace on page 9-613.

9.94 #pragma Otime on page 9-614.

7.126 --output_dir=directory_name

Specifies 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.

Example

```
armcc -c --output_dir=obj f1.c f2.c
```

Result:

```
obj/f1.o
obj/f2.o
```

Related references

7.10 --asm_dir=directory_name on page 7-280.

7.38 --depend_dir=directory_name on page 7-312.

7.99 --list_dir=directory_name on page 7-378.

7.127 -P

Preprocesses source code without compiling, but does not generate line markers in the preprocessed output.

Usage

This option can be of use when the preprocessed output is destined to be parsed by a separate script or utility.

Related references

7.53 -E on page 7-327.

7.128 --parse_templates, --no_parse_templates

Enables and 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. ARM does not recommend its use.

Mode

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

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.

Related references

7.36 --dep_name, --no_dep_name on page 7-310.

10.9 Template instantiation in ARM C++ on page 10-719.

7.129 --pch

Uses a PCH file if it exists, creates a PCH file otherwise.

———— **Note** ————

This option is deprecated.

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

————————————

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.

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.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.

Related references

7.130 --pch_dir=dir on page 7-412.
7.131 --pch_messages, --no_pch_messages on page 7-413.
7.132 --pch_verbose, --no_pch_verbose on page 7-414.
7.30 --create_pch=filename on page 7-304.
7.166 --use_pch=filename on page 7-451.
9.85 #pragma hdrstop on page 9-605.
9.90 #pragma no_pch on page 9-610.

7.130 --pch_dir=dir

Specifies the directory where PCH files are stored.

Note

This option is deprecated.

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

The directory is accessed whenever PCH files are created or used.

You can use this option with automatic or manual PCH mode.

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`.

Errors

If the specified directory *`dir`* does not exist, the compiler generates an error.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.22 Automatic Precompiled Header (PCH) file processing on page 3-90.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.

Related references

7.30 --create_pch=filename on page 7-304.
7.129 --pch on page 7-411.
7.131 --pch_messages, --no_pch_messages on page 7-413.
7.132 --pch_verbose, --no_pch_verbose on page 7-414.
7.166 --use_pch=filename on page 7-451.
9.85 #pragma hdrstop on page 9-605.
9.90 #pragma no_pch on page 9-610.

7.131 --pch_messages, --no_pch_messages

Enables and disables the display of messages indicating that a PCH file is used in the current compilation.

Note

This option is deprecated.

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

Default

The default is `--pch_messages`.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.

3.24 Precompiled Header (PCH) file creation requirements on page 3-93.

3.30 Message output during Precompiled Header (PCH) processing on page 3-100.

Related references

7.30 --create_pch=filename on page 7-304.

7.129 --pch on page 7-411.

7.130 --pch_dir=dir on page 7-412.

7.132 --pch_verbose, --no_pch_verbose on page 7-414.

7.166 --use_pch=filename on page 7-451.

9.85 #pragma hdrstop on page 9-605.

9.90 #pragma no_pch on page 9-610.

7.132 --pch_verbose, --no_pch_verbose

Enables and disables the display of messages giving reasons why a file cannot be precompiled.

Note

This option is deprecated.

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

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.

Default

The default is `--no_pch_verbose`.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.30 Message output during Precompiled Header (PCH) processing on page 3-100.

Related references

7.30 --create_pch=filename on page 7-304.
7.129 --pch on page 7-411.
7.130 --pch_dir=dir on page 7-412.
7.132 --pch_verbose, --no_pch_verbose on page 7-414.
7.131 --pch_messages, --no_pch_messages on page 7-413.
7.166 --use_pch=filename on page 7-451.
9.85 #pragma hdrstop on page 9-605.
9.90 #pragma no_pch on page 9-610.

7.133 --pending_instantiations=n

Specifies the maximum number of concurrent instantiations of a template in C++.

Syntax

`--pending_instantiations=n`

Where:

n
: is the maximum number of concurrent instantiations permitted.

If *n* is zero, there is no limit.

Mode

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

Default

If you do not specify a `--pending_instantiations` option, then the compiler assumes `--pending_instantiations=64`.

Usage

Use this option to detect runaway recursive instantiations.

7.134 --phony_targets

Emits 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`.

Example

Example output:

```
source.o: source.c
source.o: header.h
header.h:
```

Related references

7.37 --depend=filename on page 7-311.

7.39 --depend_format=string on page 7-313.

7.41 --depend_system_headers, --no_depend_system_headers on page 7-315.

7.42 --depend_target=target on page 7-316.

7.80 --ignore_missing_headers on page 7-357.

7.108 -M on page 7-387.

7.109 --md on page 7-388.

7.135 --pointer_alignment=num

Specifies unaligned pointer support required for an application.

Syntax

`--pointer_alignment=num`

Where *num* is one of:

1
: Accesses through pointers have an alignment of one, that is, byte-aligned or unaligned.

2
: Accesses through pointers have an alignment of at most two, that is, at most halfword aligned.

4
: Accesses through pointers have 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`.

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.

Restrictions

De-aligning pointers might increase the code size, even on processors 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 processors 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.

Related references

9.12 __packed on page 9-527.

9.95 #pragma pack(n) on page 9-615.

4.32 Compiler storage of data objects by natural byte alignment on page 4-144.

7.136 --preinclude=filename

Includes the source code of the specified file at the beginning of the compilation.

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`.

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.

Example

```
armcc --preinclude file1.h --preinclude file2.h -c source.c
```

Related concepts

2.9 Factors influencing how the compiler searches for header files on page 2-50.

Related references

7.90 -Jdir[,dir,...] on page 7-367.
7.79 -Idir[,dir,...] on page 7-356.
7.91 --kandr_include on page 7-368.
7.159 --sys_include on page 7-443.
2.10 Compiler command-line options and search paths on page 2-51.

7.137 --preprocess_assembly

Relaxes certain rules when producing preprocessed compiler output, to provide greater flexibility when preprocessing assembly language source code.

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
  ```

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.

Related references

7.127 -P on page 7-409.
7.18 -C on page 7-289.
7.53 -E on page 7-327.

7.138 --preprocessed

Forces the preprocessor to handle files with `.i` filename extensions as if macros have already been substituted.

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.

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.

Example

```
armcc --preprocessed foo.i -c -o foo.o
```

Related references

7.23 --compile_all_input, --no_compile_all_input on page 7-295.

7.53 -E on page 7-327.

7.139 --protect_stack, --no_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.
- 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. The compiler stores a copy of the guard variable's value at another location, and uses the copy to check that the guard has not been overwritten, indicating a buffer overflow.

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`.

Default

The default is `--no_protect_stack`.

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);
}
```

7.140 --reassociate_saturation, --no_reassociate_saturation

Enables and disables more aggressive optimization in loops that use saturating arithmetic.

Usage

Saturating addition is not associative. That is, `(x+y)+z` might not be equal to `x+(y+z)`. For example, with a saturating maximum of 50, `(40+20)-10 = 40` while `40+(20-10) = 50`.

Some compiler optimizations rely on associativity, using re-association to rearrange expressions into a more efficient sequence.

The `--no_reassociate_saturation` option prohibits re-association of saturating addition, and therefore limits the level of optimization on saturating arithmetic.

The `--reassociate_saturation` option instructs the compiler to re-associate saturating additions, and might enable optimizations when compiling with other options, such as `-O3 -Otime`.

Restriction

Saturating addition is not associative, so enabling `--reassociate_saturation` could affect the result with a reduction in accuracy.

Default

The default is `--no_reassociate_saturation`.

Examples

The following code contains the function `L_mac`, which performs saturating additions.

```
#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;
}
```

7.141 --reduce_paths, --no_reduce_paths

Enables and 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.

Mode

This option is effective on Windows systems only.

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

ARM recommends that you avoid using long and deeply nested file paths, in preference to minimizing path lengths using the `--reduce_paths` option.

Default

The default is `--no_reduce_paths`.

Example

Compiling the file

```
..\..\..\xyzzy\xyzzy\objects\file.c
```

from the directory

```
\foo\bar\baz\gazonk\quux\bop
```

results in an actual path of

```
\foo\bar\baz\gazonk\quux\bop\..\..\..\xyzzy\xyzzy\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\xyzzy\xyzzy\objects\file.c
```

7.142 --relaxed_ref_def, --no_relaxed_ref_def

Permits multiple object files to use tentative definitions of global variables.

Some traditional programs are written using this declaration style.

Usage

This option is primarily provided for compatibility with GNU C. ARM does not recommend using this option for new application code.

Default

The default is strict references and definitions. (Each global variable can only be declared in one object file.)

Restrictions

This option is not available in C++.

7.143 --remarks

Enables the display of remark messages, including any messages redesignated to remark severity using `--diag_remark`.

Note

The compiler does not issue remarks by default.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

7.144 --remove_unneeded_entities, --no_remove_unneeded_entities

Controls whether debug information is generated for all source symbols, or only for those source symbols actually used.

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.

Restrictions

The effects of these options are restricted to debug information.

Default

The default is `--no_remove_unneeded_entities`.

Related information

The DWARF Debugging Standard, http://dwarfstd.org/.

7.145 --restrict, --no_restrict

Enables and 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.

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.

Related references

8.13 restrict on page 8-478.

7.146 --retain=option

Restricts the optimizations performed by the compiler.

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 `a||b` transformed to `a|b`. This supports *Modified Condition Decision Coverage* (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.

`rwdata`
prevents read-write data being removed.

`data:order`
prevents data being reordered.

If *`option`* is unspecified, the compiler faults use of `--retain`.

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.

Related references

9.40 __attribute__((nomerge)) function attribute on page 9-559.
9.43 __attribute__((notailcall)) function attribute on page 9-562.

7.147 --rtti, --no_rtti

Controls support for the RTTI features `dynamic_cast` and `typeid` in C++.

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.

Mode

These options are effective only if the source language is C++.

Default

The default is `--rtti`.

Related references

7.148 --rtti_data, --no_rtti_data on page 7-430.

7.148 --rtti_data, --no_rtti_data

Enables and disables the generation of C++ RTTI data.

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.

Mode

These options are effective only if the source language is C++.

Default

The default is `--rtti_data`.

Related references

7.58 --exceptions, --no_exceptions on page 7-332.

7.147 --rtti, --no_rtti on page 7-429.

7.149 -S

Outputs 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.

Related references

7.6 --apcs=qualifier...qualifier on page 7-273.

7.9 --asm on page 7-279.

7.17 -c on page 7-288.

7.85 --info=totals on page 7-362.

7.89 --interleave on page 7-366.

7.98 --list on page 7-376.

7.118 -o filename on page 7-397.

Related information

armasm User Guide.

7.150 --share_inlineable_strings, --no_share_inlineable_strings

Controls whether multiple instances of the same inlined string literal use the same object.

With the `--share_inlineable_strings` option, all instances of an inlined string literal use the same object. Specifically:

- String literals in the same translation unit are the same object in that translation unit.
- String literals in externally-visible inline functions are the same object in all translation units.

This can also cause different string literals with the same value to be the same object.

The `--no_share_inlineable_strings` option suppresses this behavior. That is, the compiler only shares string literals if it provides a performance or code size benefit. For example, consider a string literal that is out of range of an `ADR` instruction. In this case, the address must be loaded from a literal pool, which costs 4 bytes and is slower than an `ADR` instruction. As a result, with `-Otime` the compiler would not share any strings that were out of range, and with `-Ospace` the compiler would not share any strings smaller than 4 bytes.

Note

The `--share_inlineable_strings` and `--no_share_inlineable_strings` options affect string literals in:

- C or C++ functions not declared inline, but which the compiler has chosen to inline.
- C inline functions.

The behavior of strings in C++ inline functions is separate from these options, and is defined by the *C++ ABI*.

Example

Consider the following example code:

```
#include <stdio.h>

extern inline char *getString(void) { return "abc"; }

int main(int argc, char **argv)
{
  char *a = getString();
  char *(*ptr)() = getString;
  char *b = ptr();


  int a_addr=(int)a;
  int b_addr=(int)b;


  printf("String \"%s\" from getString() called directly is an object at address:          %d
\n", a, a_addr);
  printf("String \"%s\" from getString() called using a pointer is an object at address:   %d
\n", b, b_addr);

  if (a_addr == b_addr) {
    printf("Objects are the same.\n");
  } else {
    printf("Objects are different.\n");
  }
}
```

By default (that is, with the `--share_inlineable_strings` option) both instances of the string literal use a single, shared, object:

```
armcc --share_inlineable_strings  --c99 test.c  -o-
```

Running the compiled image produces the following output:

```
String "abc" from getString() called directly is an object at address:         36812
String "abc" from getString() called using a pointer is an object at address:   36812
Objects are the same.
```

Compiling the same code with the `--no_share_inlineable_strings` option results in multiple string objects:

```
armcc --no_share_inlineable_strings  --c99 test.c  -o-
```

Running the compiled image produces the following output:

```
String "abc" from getString() called directly is an object at address:         33004
String "abc" from getString() called using a pointer is an object at address:   36616
Objects are different.
```

Related references

7.124 -Ospace on page 7-406.

7.125 -Otime on page 7-407.

7.151 --show_cmdline

Outputs the command line used by the compiler.

Usage

Shows the command line after processing by the compiler, and can be useful to check:

- 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 error stream (`stderr`).

Related references

7.78 --help on page 7-355.

7.1 -Aopt on page 7-268.

7.54 --echo on page 7-328.

7.92 -Lopt on page 7-369.

7.170 --via=filename on page 7-455.

7.152 --signed_bitfields, --unsigned_bitfields

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.

Default

The default is `--unsigned_bitfields`.

Note

The AAPCS requirement for bitfields to default to unsigned on ARM, is relaxed in version 2.03 of the standard.

Example

```
typedef int integer;
struct
{
    integer x : 1;
} bf;
```

Compiling this code with `--signed_bitfields` causes `x` to be treated as a signed bitfield.

Related information

Procedure Call Standard for the ARM Architecture.

7.153 --signed_chars, --unsigned_chars

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.

————————————

Default

The default is `--unsigned_chars`.

Related references

9.158 Predefined macros on page 9-697.

7.154 --split_ldm

Splits LDM and STM instructions performing large numbers of register transfers into multiple LDM or STM instructions, to help reduce interrupt latency on some ARM systems.

When `--split_ldm` is selected, the maximum number of register transfers for an `LDM` or `STM` instruction is limited to:

- Five, for all `STM`s.
- Five, for `LDM`s that do not load the PC.
- Four, for `LDM`s that load the PC.

Where register transfers beyond these limits are required, multiple `LDM` or `STM` instructions are used.

Usage

The `--split_ldm` option can 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.

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.

Related concepts

6.16 Inline assembler and instruction expansion in C and C++ code on page 6-232.

7.155 --split_sections

Generates 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, ARM recommends 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.

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.

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
```

Related references

7.32 --data_reorder, --no_data_reorder on page 7-306.
7.62 --feedback=filename on page 7-336.
7.114 --multifile, --no_multifile on page 7-393.
9.47 __attribute__((section("name"))) function attribute on page 9-566.
9.77 #pragma arm section [section_type_list] on page 9-596.
2.14 Linker feedback during compilation on page 2-55.

7.156 --strict, --no_strict

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.

Default

The default is `--no_strict`.

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++.

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 using the `--diag_error`, `--diag_remark`, and `--diag_warning` options.

Example

```
void foo(void)
{
    long long i; /* okay in nonstrict C90 */
}
```

Compiling this code with `--strict` generates an error.

Related references

7.5 --anachronisms, --no_anachronisms on page 7-272.
7.19 --c90 on page 7-290.
7.20 --c99 on page 7-291.
7.73 --gnu on page 7-350.
7.25 --cpp on page 7-297.
7.26 --cpp11 on page 7-298.
7.27 --cpp_compat on page 7-299.
1.2 Source language modes of the compiler on page 1-29.
2.7 Filename suffixes recognized by the compiler on page 2-47.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.157 --strict_warnings on page 7-440.
8.19 Dollar signs in identifiers on page 8-484.

7.157 --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.

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 using the `--diag_error`, `--diag_remark`, and `--diag_warning` options.

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.

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.

Related references

7.5 --anachronisms, --no_anachronisms on page 7-272.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.157 --strict_warnings on page 7-440.
8.19 Dollar signs in identifiers on page 8-484.
1.2 Source language modes of the compiler on page 1-29.

7.158 --string_literal_pools, --no_string_literal_pools

Controls whether the compiler places string constants in literal pools.

With the `--string_literal_pools` option, where there are string literals in source code, the compiler usually places the character data in a literal pool:

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 32 bytes (alignment 4)
    Address: 0x00000000

    $a
    .text
    main
        0x00000000:    e92d4010    .@-.    PUSH     {r4,lr}
        0x00000004:    e28f0008    ....    ADR      r0,{pc}+0x10 ; 0x14
        0x00000008:    ebfffffe    ....    BL       puts
        0x0000000c:    e3a00000    ....    MOV      r0,#0
        0x00000010:    e8bd8010    ....    POP      {r4,pc}
    $d
        0x00000014:    6c6c6548    Hell    DCD    1819043144
        0x00000018:    6f77206f    o wo    DCD    1870078063
        0x0000001c:    00646c72    rld.    DCD    6581362
```

The `--no_string_literal_pools` option instructs the compiler to place string constants in a separate `.conststring` or `.constdata` section, and load the address of the character data from an integer literal pool, as follows:

```
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 24 bytes (alignment 4)
    Address: 0x00000000

    $a
    .text
    main
        0x00000000:    e59f000c    ....    LDR      r0,[pc,#12] ; [0x14] = 0
        0x00000004:    e92d4010    .@-.    PUSH     {r4,lr}
        0x00000008:    ebfffffe    ....    BL       puts
        0x0000000c:    e3a00000    ....    MOV      r0,#0
        0x00000010:    e8bd8010    ....    POP      {r4,pc}
    $d
        0x00000014:    00000000    ....    DCD    0

** Section #4 '.conststring' (SHT_PROGBITS) [SHF_ALLOC + SHF_MERGE + SHF_STRINGS]
    Size   : 12 bytes (alignment 4)
    Address: 0x00000000

    0x000000:   48 65 6c 6c 6f 20 77 6f 72 6c 64 00                Hello world.
```

If you also specify the `--no_integer_literal_pools` option, the compiler constructs the address of the character data with a pair of `MOVW`/`MOVT` instructions.

Default

The default is `--string_literal_pools`.

`--execute_only` implies `--no_string_literal_pools`, unless `--string_literal_pools` is explicitly specified.

Note

Do not use `--execute_only` in conjunction with `--string_literal_pools`. If you do, then the compiler places the literal pool in an unreadable, execute-only code region.

Related concepts

3.19 Compiler support for literal pools on page 3-86.

Related references

7.87 --integer_literal_pools, --no_integer_literal_pools on page 7-364.
7.14 --branch_tables, --no_branch_tables on page 7-284.

7.63 --float_literal_pools, --no_float_literal_pools on page 7-337.

7.60 --execute_only on page 7-334.

7.159 --sys_include

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.

Related concepts

2.9 Factors influencing how the compiler searches for header files on page 2-50.

Related references

7.90 -Jdir[,dir,...] on page 7-367.

7.79 -Idir[,dir,...] on page 7-356.

7.91 --kandr_include on page 7-368.

7.136 --preinclude=filename on page 7-418.

2.10 Compiler command-line options and search paths on page 2-51.

7.160 --thumb

Targets the Thumb instruction set.

Default

This is the default option for targets that do not support the ARM instruction set.

Related tasks

4.4 Selecting the target processor at compile time on page 4-111.

Related references

7.7 --arm on page 7-277.

9.76 #pragma arm on page 9-595.

9.99 #pragma thumb on page 9-620.

Related information

ARM architectures supported by the toolchain.

7.161 --trigraphs, --no_trigraphs

Enables and disables trigraph recognition.

Default

The default is `--trigraphs`, except in GNU mode, where the default is `--no_trigraphs`.

Related information

ISO/IEC 9899:TC2.

7.162 --type_traits_helpers, --no_type_traits_helpers

Enables and disables 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.

Related references

7.76 --gnu_version=version on page 7-353.

7.163 -Uname

Removes any initial definition of the specified macro.

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.

————————————

Syntax

`-Uname`

Where:

`name`

is the name of the macro to be undefined.

Usage

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

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`.

Related references

7.18 -C on page 7-289.

7.31 -Dname[(parm-list)][=def] on page 7-305.

7.53 -E on page 7-327.

7.108 -M on page 7-387.

9.158 Predefined macros on page 9-697.

7.164 --unaligned_access, --no_unaligned_access

Enables and disables unaligned accesses to data on ARM architecture-based processors.

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-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.

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 in ARMv6, except ARMv6-M, 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 processor. The MMU must be on, and the memory marked as normal memory.

ARMv6-M faults all unaligned data accesses.

To enable unaligned support in ARMv7:

- In ARMv7-R, clear the `A` bit, bit 1, in the *System Control Register* (SCTLR).
- In ARMv7-M, clear the `UNALIGN_TRP` bit, bit 3, in the *Configuration and Control Register* (CCR).

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 and ARMv7 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 processor.

To enable alignment fault checking in ARMv7:
- In ARMv7-R, set the `A` bit, bit 1, in the SCTLR.
- In ARMv7-M, set the `UNALIGN_TRP` bit, bit 3, in the CCR.

——— **Note** ———

ARM processors do not provide support for unaligned doubleword accesses, for example unaligned accesses to **`long long`** integers. Doubleword accesses must be either eight-byte or four-byte aligned.

———————————

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.

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.

Related references

7.29 --cpu=name compiler option on page 7-302.

Related information

--unaligned_access, --no_unaligned_access assembler option.

7.165 --use_frame_pointer, --no_use_frame_pointer

Sets the frame pointer to the current stack frame.

Using the `--use_frame_pointer` option reserves a register to store the frame pointer.

For newer processors that support Thumb-2 technology (ARMv6T2 and later), the reserved register is always `R11`.

For older processors that do not support Thumb-2 technology, the reserved register is `R11` in ARM code and `R7` in Thumb code.

Default

The default is `--no_use_frame_pointer`. That is, register `R11` (or register `R7` for Thumb code on older processors) is available for use as a general-purpose register.

Related information

ARM registers.

General-purpose registers.

7.166 --use_pch=filename

Uses the specified PCH file as part of the current compilation.

——— **Note** ———

This option is deprecated.

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

———————

This option takes precedence if you include `--pch` on the same command line.

Syntax

`--use_pch=`*`filename`*

Where:

`filename`
is the PCH file to be used as part of the current compilation.

Restrictions

The effect of this option is negated if you include `--create_pch=`*`filename`* on the same command line.

Errors

If the specified file does not exist, or is not a valid PCH file, the compiler generates an error.

Related concepts

3.27 Manually specifying the filename and location of a Precompiled Header (PCH) file on page 3-97.

3.21 Precompiled Header (PCH) files on page 3-88.

Related references

7.30 --create_pch=filename on page 7-304.

7.129 --pch on page 7-411.

7.130 --pch_dir=dir on page 7-412.

7.131 --pch_messages, --no_pch_messages on page 7-413.

7.132 --pch_verbose, --no_pch_verbose on page 7-414.

9.85 #pragma hdrstop on page 9-605.

9.90 #pragma no_pch on page 9-610.

7.167 --using_std, --no_using_std

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. ARM does not recommend its use.

Mode

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

Default

The default is `--no_using_std`.

Related references

10.10 Namespaces in ARM C++ on page 10-720.

7.168 --version_number

Displays the version of `armcc` you are using.

Usage

The compiler displays the version number in the format `nnnbbbb`, where:

- `nnn` is the version number.
- `bbbb` is the build number.

Example

Version 5.06 build 0019 is displayed as `5060019`.

Related references

7.172 --vsn on page 7-457.

7.78 --help on page 7-355.

7.169 --vfe, --no_vfe

Enables and 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.

Mode

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

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.

Related references

16.2 Calling a pure virtual function on page 16-855.

Related information

Elimination of unused virtual functions.

7.170 --via=filename

Reads an additional list of input filenames and compiler options from *filename*.

Syntax

`--via=`*`filename`*

Where *filename* is the name of a via file containing options to be included on the command line.

Usage

You can enter multiple `--via` options on the compiler command line. The `--via` options can also be included within a via file.

Example

Given a source file `main.c`, a via file `apcs.txt` containing the line:

```
--apcs=/rwpi --no_lower_rwpi --via=L_apcs.txt
```

and a second via file `L_apcs.txt` containing the line:

```
-L--rwpi -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_rwpi --apcs=/rwpi -L--rwpi -L--callgraph -L-o"main.axf" main.c
```

Related references

13.2 Via file syntax rules on page 13-827.

Related information

Methods of specifying command-line options.

7.171 --vla, --no_vla

Enables or disables support for variable length arrays.

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.

Note

Memory for variable length arrays is allocated at runtime, on the heap.

Example

```
size_t arr_size(int n)
{
    char array[n];          // variable length array, dynamically allocated
    return sizeof array;   // evaluated at runtime
}
```

Related references

7.19 --c90 on page 7-290.
7.20 --c99 on page 7-291.
7.25 --cpp on page 7-297.
7.73 --gnu on page 7-350.

7.172 --vsn

Displays the version information and the license details.

Example

```
> armcc --vsn
Product: ARM Compiler N.nn
Component: ARM Compiler N.nn (toolchain_build_number)
Tool: armcc [build_number]
license_type
Software supplied by: ARM Limited
```

Related references

7.78 --help on page 7-355.

7.168 --version_number on page 7-453.

7.173 -W

Suppresses all warning messages.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

7.174 --wchar, --no_wchar

Permits or forbids the use of wchar_t.

It does not necessarily fault declarations, providing they are unused.

Usage

Use this option to create an object file that is independent of `wchar_t` size.

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.

Default

The default is `--wchar`.

Related references

7.175 --wchar16 on page 7-460.

7.176 --wchar32 on page 7-461.

7.175 --wchar16

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`.

Default

The compiler assumes `--wchar16` unless `--wchar32` is explicitly specified.

Related references

7.174 --wchar, --no_wchar on page 7-459.

7.176 --wchar32 on page 7-461.

9.158 Predefined macros on page 9-697.

7.176 --wchar32

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`.

Default

The compiler assumes `--wchar16` unless `--wchar32` is explicitly specified.

Related references

9.158 Predefined macros on page 9-697.

7.175 --wchar16 on page 7-460.

7.174 --wchar, --no_wchar on page 7-459.

7.74 --gnu_defaults on page 7-351.

7.177 --whole_program

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.

Usage

Use this option to gain maximum performance from a small program.

Restriction

Do not use this option if you do not have all of the source code to give to the compiler.

Related references

7.114 --multifile, --no_multifile on page 7-393.

7.178 --wrap_diagnostics, --no_wrap_diagnostics

Enables and disables the wrapping of error message text when it is too long to fit on a single line.

Default

The default is `--no_wrap_diagnostics`.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

Chapter 8
Language Extensions

Describes the language extensions that the compiler supports.

It contains the following sections:

- *8.1 Preprocessor extensions* on page 8-466.
- *8.2 #assert* on page 8-467.
- *8.3 #include_next* on page 8-468.
- *8.4 #unassert* on page 8-469.
- *8.5 #warning* on page 8-470.
- *8.6 C99 language features available in C90* on page 8-471.
- *8.7 // comments* on page 8-472.
- *8.8 Subscripting struct* on page 8-473.
- *8.9 Flexible array members* on page 8-474.
- *8.10 C99 language features available in C++ and C90* on page 8-475.
- *8.11 Variadic macros* on page 8-476.
- *8.12 long long* on page 8-477.
- *8.13 restrict* on page 8-478.
- *8.14 Hexadecimal floats* on page 8-479.
- *8.15 Standard C language extensions* on page 8-480.
- *8.16 Constant expressions* on page 8-481.
- *8.17 Array and pointer extensions* on page 8-482.
- *8.18 Block scope function declarations* on page 8-483.
- *8.19 Dollar signs in identifiers* on page 8-484.
- *8.20 Top-level declarations* on page 8-485.
- *8.21 Benign redeclarations* on page 8-486.
- *8.22 External entities* on page 8-487.
- *8.23 Function prototypes* on page 8-488.

- *8.24 Standard C++ language extensions* on page 8-489.
- *8.25 ? operator* on page 8-490.
- *8.26 Declaration of a class member* on page 8-491.
- *8.27 friend* on page 8-492.
- *8.28 Read/write constants* on page 8-493.
- *8.29 Scalar type constants* on page 8-494.
- *8.30 Specialization of nonmember function templates* on page 8-495.
- *8.31 Type conversions* on page 8-496.
- *8.32 Standard C and Standard C++ language extensions* on page 8-497.
- *8.33 Address of a register variable* on page 8-498.
- *8.34 Arguments to functions* on page 8-499.
- *8.35 Anonymous classes, structures and unions* on page 8-500.
- *8.36 Assembler labels* on page 8-501.
- *8.37 Empty declaration* on page 8-502.
- *8.38 Hexadecimal floating-point constants* on page 8-503.
- *8.39 Incomplete enums* on page 8-504.
- *8.40 Integral type extensions* on page 8-505.
- *8.41 Label definitions* on page 8-506.
- *8.42 Long float* on page 8-507.
- *8.43 Nonstatic local variables* on page 8-508.
- *8.44 Structure, union, enum, and bitfield extensions* on page 8-509.
- *8.45 GNU extensions to the C and C++ languages* on page 8-510.

8.1 Preprocessor extensions

The compiler supports several extensions to the preprocessor, including the `#assert` preprocessing extensions of System V release 4.

Related references

8.2 #assert on page 8-467.

8.3 #include_next on page 8-468.

8.4 #unassert on page 8-469.

8.5 #warning on page 8-470.

8.2 #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.

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`*.

Usage

You can test a predicate name defined using `#assert` 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 predicate can have multiple values. That is, subsequent assertions do not override preceding assertions.

Example

The following example assigns multiple values and shows the results `#if` expressions:

```
#assert foo(one)      // Assigns the value "one"
#assert foo(two)      // Assigns the value "two"
#assert foo(three)    // Assigns the value "three"
#unassert foo(two)    // Unassigns the value "two"

#if #foo(one)...      // 1
#if #foo(two)...      // 0, because of #unassert
#if #foo(three)...    // 1
#if #foo(four)...     // 0, because this value was never asserted
```

Related references

8.4 #unassert on page 8-469.

8.3 #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.

————————————

8.4 #unassert

You can delete a predicate name using the `#unassert` preprocessing directive.

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.

Related references

8.2 #assert on page 8-467.

8.5 #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.

Restrictions

The `#warning` directive is not available if the `--strict` option is specified. If used, it produces an error.

Related references

7.156 --strict, --no_strict on page 7-439.

8.6 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.

Related references

8.7 // comments on page 8-472.

8.8 Subscripting struct on page 8-473.

8.9 Flexible array members on page 8-474.

8.7 // 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.

Example

```
// this is a comment
```

Related concepts

4.59 New language features of C99 on page 4-180.

8.8 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.

Example

```
struct Subscripting_Struct
{
    int a[4];
};
extern struct Subscripting_Struct Subscripting_0(void);
int Subscripting_1 (int index)
{
    return Subscripting_0().a[index];
}
```

8.9 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.

Example

```
typedef struct
{
    int len;
    char p[]; // incomplete array type, for use in a malloc’d data structure
} str;
```

Related concepts

4.59 New language features of C99 on page 4-180.

8.10 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.

———————

Related references

8.11 Variadic macros on page 8-476.

8.12 long long on page 8-477.

8.13 restrict on page 8-478.

8.14 Hexadecimal floats on page 8-479.

8.11 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.

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);
}
```

Related concepts

4.59 New language features of C99 on page 4-180.

Related references

7.73 --gnu on page 7-350.

8.12 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.

Related references

9.9 __int64 on page 9-524.

8.13 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.

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++.

Restrictions

The declaration of intent is effectively a promise to the compiler that, if broken, results in undefined behavior.

Examples

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)
        *q++ = *s++;
}
```

Related concepts

4.59 New language features of C99 on page 4-180.

Related references

7.145 --restrict, --no_restrict on page 7-427.

8.14 Hexadecimal floats

C90 and C++ support floating-point numbers that can be written in hexadecimal format.

Example

```
float hex_floats(void)
{
    return 0x1.fp3;    // 1.55e1
}
```

Related concepts

4.59 New language features of C99 on page 4-180.

8.15 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++.

Related references

8.16 Constant expressions on page 8-481.

8.17 Array and pointer extensions on page 8-482.

8.18 Block scope function declarations on page 8-483.

8.19 Dollar signs in identifiers on page 8-484.

8.20 Top-level declarations on page 8-485.

8.21 Benign redeclarations on page 8-486.

8.22 External entities on page 8-487.

8.23 Function prototypes on page 8-488.

8.16 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.

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.

Example 2, constant value initializers

The following table 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 8-1 Behavior of constant value initializers in comparison with ISO Standard C

Example source code	ISO C Standard	ARM compilation tools	
		`--strict` mode	Nonstrict mode
`extern int const c = 10;`	Valid	Valid	Valid
`extern int const x = c + 10;`	Not valid	Not valid	Valid
`static int y = c + 10;`	Not valid	Not valid	Valid
`static int const z = c + 10;`	Not valid	Not valid	Valid
`extern int *const cp = (int*)0x100;`	Valid	Valid	Valid
`extern int *const xp = cp + 0x100;`	Not valid	Not valid	Valid
`static int *yp = cp + 0x100;`	Not valid	Not valid	Valid
`static int *const zp = cp + 0x100;`	Not valid	Not valid	Valid

Related references

7.61 --extended_initializers, --no_extended_initializers on page 7-335.

7.156 --strict, --no_strict on page 7-439.

7.157 --strict_warnings on page 7-440.

8.17 Array and pointer extensions

The compiler supports a number of array and pointer extensions, for example permitting assignment between pointers to types that are interchangeable but not identical.

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.

8.18 Block scope function declarations

The compiler supports the following extensions to block scope function declarations.

- 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 static linkage by default.

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 */
```

8.19 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.

Example

```
#define DOLLAR$
```

Related references

7.50 --dollar, --no_dollar on page 7-324.
7.156 --strict, --no_strict on page 7-439.

8.20 Top-level declarations

A C input file can contain no top-level declarations.

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`.

————————————

Related references

7.143 --remarks on page 7-425.

8.21 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.

Example

```
typedef int INT;
typedef int INT; /* redeclaration */
```

8.22 External entities

External entities declared in other scopes are visible.

Errors

The compiler generates a warning if an external entity declared in another scope is visible.

Example

```
void f1(void)
{
    extern void f();
}
void f2(void)
{
    f(); /* Out of scope declaration */
}
```

8.23 Function prototypes

The compiler recognizes function prototypes that override old-style nonprototype definitions that appear at a later position in your code.

Errors

The compiler generates a warning message if you use old-style function prototypes.

Example

```
int function_prototypes(char);
// Old-style function definition.
int function_prototypes(x)
    char x;
{
    return x == 0;
}
```

8.24 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.

Related references

8.25 ? operator on page 8-490.

8.26 Declaration of a class member on page 8-491.

8.27 friend on page 8-492.

8.28 Read/write constants on page 8-493.

8.29 Scalar type constants on page 8-494.

8.30 Specialization of nonmember function templates on page 8-495.

8.31 Type conversions on page 8-496.

8.25 ? 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.

Example

```
char *p = x ? "abc" : "def";
```

8.26 Declaration of a class member

A qualified name can be used in the declaration of a class member.

Errors

A warning is issued if a qualified name is used in the declaration of a class member.

Example

```
struct A
{
    int A::f();  // is the same as int f();
};
```

8.27 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.

Example

```
class B;
class A
{
    friend B;  // is the same as "friend class B"
};
```

Related references

7.156 --strict, --no_strict on page 7-439.

8.28 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 /rwpi`. 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 /rwpi` option deviates from the ISO C++ Standard. The declarations in this example assume that `x` is in a read-only segment:

```
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 this example 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.

```
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 the following example produces a compile 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.

Related references

7.6 --apcs=qualifier...qualifier on page 7-273.

8.29 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.

Errors

A warning is issued if you define a member of constant integral type within a class.

Example

```
class A
{
    const int size = 10; // must be static const int size = 10;
    int a[size];
};
```

8.30 Specialization of nonmember function templates

As an extension, it is permitted to specify a storage class on a specialization of a nonmember function template.

8.31 Type conversions

Type conversion between a pointer to an `extern "C"` function and a pointer to an `extern "C++"` function is permitted.

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
```

8.32 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.

Related references

8.33 Address of a register variable on page 8-498.

8.34 Arguments to functions on page 8-499.

8.35 Anonymous classes, structures and unions on page 8-500.

8.36 Assembler labels on page 8-501.

8.37 Empty declaration on page 8-502.

8.38 Hexadecimal floating-point constants on page 8-503.

8.39 Incomplete enums on page 8-504.

8.40 Integral type extensions on page 8-505.

8.41 Label definitions on page 8-506.

8.42 Long float on page 8-507.

8.43 Nonstatic local variables on page 8-508.

8.44 Structure, union, enum, and bitfield extensions on page 8-509.

8.33 Address of a register variable

The address of a variable with **`register`** storage class can be taken.

Errors

The compiler generates a warning if you take the address of a variable with **`register`** storage class.

Example

```
void foo(void)
{
    register int i;
    int *j = &i;
}
```

8.34 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.

8.35 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;
}
```

Related references

9.75 #pragma anon_unions, #pragma no_anon_unions on page 9-594.

8.36 Assembler labels

Assembly labels specify the assembly code name to use for a C symbol.

For example, you might have assembly 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`.

Related references

9.5 __asm on page 9-519.

8.37 Empty declaration

An empty declaration, that is a semicolon with nothing before it, is permitted.

Example

```
; // do nothing
```

8.38 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.

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.

8.39 Incomplete enums

Forward declarations of `enums` are supported.

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 };
```

8.40 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.

8.41 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.

Errors

The compiler generates a warning if a label definition is followed immediately by a right brace.

Example

```
void foo(char *p)
{
    if (p)
    {
        /* ... */
label:
    }
}
```

8.42 Long float

`long float` is accepted as a synonym for **`double`**.

8.43 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.

8.44 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 required, 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 initializes 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 */
```

- In C, oversized bitfields are supported. Oversized bitfields are part of standard C++. The semantics of oversized bitfields in ARM C is the same as for standard C++.

  An oversized bitfield is a field in a structure which has the form *`basetype v`*`:`*`N`* or *`basetype`*`:`*`N`* where the size in bits of *`basetype`* is less than *`N`*. For example, in `char a:16;` type `char` has 8 bits while the bitfield has 16 bits. The extra bits are treated as padding.
- Bitfields can have base types that are **`enum`** types or integral types besides **`int`** and **`unsigned int`**.

Related concepts

4.59 New language features of C99 on page 4-180.

Related references

9.74 Pragmas on page 9-593.

8.45 GNU extensions to the C and C++ languages

GNU provides many extensions to the C and C++ languages, and the ARM compiler supports many of these extensions. 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.

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).

Related references

7.73 --gnu on page 7-350.

1.4 Language compliance on page 1-32.

2.7 Filename suffixes recognized by the compiler on page 2-47.

14.1 Supported GNU extensions on page 14-829.

Related information

Which GNU language extensions are supported by the ARM Compiler?.

Chapter 9
Compiler-specific Features

Describes compiler-specific features including ARM extensions to the C and C++ Standards, ARM-specific pragmas and intrinsics, and predefined macros.

It contains the following sections:

- *9.1 Keywords and operators* on page 9-515.
- *9.2 __align* on page 9-516.
- *9.3 __ALIGNOF__* on page 9-517.
- *9.4 __alignof__* on page 9-518.
- *9.5 __asm* on page 9-519.
- *9.6 __forceinline* on page 9-520.
- *9.7 __global_reg* on page 9-521.
- *9.8 __inline* on page 9-523.
- *9.9 __int64* on page 9-524.
- *9.10 __INTADDR__* on page 9-525.
- *9.11 __irq* on page 9-526.
- *9.12 __packed* on page 9-527.
- *9.13 __pure* on page 9-529.
- *9.14 __smc* on page 9-530.
- *9.15 __softfp* on page 9-531.
- *9.16 __svc* on page 9-532.
- *9.17 __svc_indirect* on page 9-533.
- *9.18 __svc_indirect_r7* on page 9-534.
- *9.19 __value_in_regs* on page 9-535.
- *9.20 __weak* on page 9-536.
- *9.21 __writeonly* on page 9-538.
- *9.22 __declspec attributes* on page 9-539.

- *9.23 __declspec(noinline)* on page 9-540.
- *9.24 __declspec(noreturn)* on page 9-541.
- *9.25 __declspec(nothrow)* on page 9-542.
- *9.26 __declspec(notshared)* on page 9-543.
- *9.27 __declspec(thread)* on page 9-544.
- *9.28 Function attributes* on page 9-545.
- *9.29 __attribute__((alias)) function attribute* on page 9-547.
- *9.30 __attribute__((always_inline)) function attribute* on page 9-549.
- *9.31 __attribute__((const)) function attribute* on page 9-550.
- *9.32 __attribute__((constructor[(priority)])) function attribute* on page 9-551.
- *9.33 __attribute__((deprecated)) function attribute* on page 9-552.
- *9.34 __attribute__((destructor[(priority)])) function attribute* on page 9-553.
- *9.35 __attribute__((format)) function attribute* on page 9-554.
- *9.36 __attribute__((format_arg(string-index))) function attribute* on page 9-555.
- *9.37 __attribute__((malloc)) function attribute* on page 9-556.
- *9.38 __attribute__((noinline)) function attribute* on page 9-557.
- *9.39 __attribute__((no_instrument_function)) function attribute* on page 9-558.
- *9.40 __attribute__((nomerge)) function attribute* on page 9-559.
- *9.41 __attribute__((nonnull)) function attribute* on page 9-560.
- *9.42 __attribute__((noreturn)) function attribute* on page 9-561.
- *9.43 __attribute__((notailcall)) function attribute* on page 9-562.
- *9.44 __attribute__((nothrow)) function attribute* on page 9-563.
- *9.45 __attribute__((pcs("calling_convention"))) function attribute* on page 9-564.
- *9.46 __attribute__((pure)) function attribute* on page 9-565.
- *9.47 __attribute__((section("name"))) function attribute* on page 9-566.
- *9.48 __attribute__((sentinel)) function attribute* on page 9-567.
- *9.49 __attribute__((unused)) function attribute* on page 9-568.
- *9.50 __attribute__((used)) function attribute* on page 9-569.
- *9.51 __attribute__((visibility("visibility_type"))) function attribute* on page 9-570.
- *9.52 __attribute__((warn_unused_result))* on page 9-571.
- *9.53 __attribute__((weak)) function attribute* on page 9-572.
- *9.54 __attribute__((weakref("target"))) function attribute* on page 9-573.
- *9.55 Type attributes* on page 9-574.
- *9.56 __attribute__((bitband)) type attribute* on page 9-575.
- *9.57 __attribute__((aligned)) type attribute* on page 9-576.
- *9.58 __attribute__((packed)) type attribute* on page 9-577.
- *9.59 __attribute__((transparent_union)) type attribute* on page 9-578.
- *9.60 Variable attributes* on page 9-579.
- *9.61 __attribute__((alias)) variable attribute* on page 9-580.
- *9.62 __attribute__((at(address))) variable attribute* on page 9-581.
- *9.63 __attribute__((aligned)) variable attribute* on page 9-582.
- *9.64 __attribute__((deprecated)) variable attribute* on page 9-583.
- *9.65 __attribute__((noinline)) constant variable attribute* on page 9-584.
- *9.66 __attribute__((packed)) variable attribute* on page 9-585.
- *9.67 __attribute__((section("name"))) variable attribute* on page 9-586.
- *9.68 __attribute__((unused)) variable attribute* on page 9-587.
- *9.69 __attribute__((used)) variable attribute* on page 9-588.
- *9.70 __attribute__((visibility("visibility_type"))) variable attribute* on page 9-589.
- *9.71 __attribute__((weak)) variable attribute* on page 9-590.
- *9.72 __attribute__((weakref("target"))) variable attribute* on page 9-591.
- *9.73 __attribute__((zero_init)) variable attribute* on page 9-592.
- *9.74 Pragmas* on page 9-593.
- *9.75 #pragma anon_unions, #pragma no_anon_unions* on page 9-594.
- *9.76 #pragma arm* on page 9-595.
- *9.77 #pragma arm section [section_type_list]* on page 9-596.
- *9.78 #pragma diag_default tag[,tag,...]* on page 9-598.

- *9.79 #pragma diag_error tag[,tag,...]* on page 9-599.
- *9.80 #pragma diag_remark tag[,tag,...]* on page 9-600.
- *9.81 #pragma diag_suppress tag[,tag,...]* on page 9-601.
- *9.82 #pragma diag_warning tag[, tag, ...]* on page 9-602.
- *9.83 #pragma exceptions_unwind, #pragma no_exceptions_unwind* on page 9-603.
- *9.84 #pragma GCC system_header* on page 9-604.
- *9.85 #pragma hdrstop* on page 9-605.
- *9.86 #pragma import symbol_name* on page 9-606.
- *9.87 #pragma import(__use_full_stdio)* on page 9-607.
- *9.88 #pragma import(__use_smaller_memcpy)* on page 9-608.
- *9.89 #pragma inline, #pragma no_inline* on page 9-609.
- *9.90 #pragma no_pch* on page 9-610.
- *9.91 #pragma Onum* on page 9-611.
- *9.92 #pragma once* on page 9-612.
- *9.93 #pragma Ospace* on page 9-613.
- *9.94 #pragma Otime* on page 9-614.
- *9.95 #pragma pack(n)* on page 9-615.
- *9.96 #pragma pop* on page 9-617.
- *9.97 #pragma push* on page 9-618.
- *9.98 #pragma softfp_linkage, #pragma no_softfp_linkage* on page 9-619.
- *9.99 #pragma thumb* on page 9-620.
- *9.100 #pragma unroll [(n)]* on page 9-621.
- *9.101 #pragma unroll_completely* on page 9-623.
- *9.102 #pragma weak symbol, #pragma weak symbol1 = symbol2* on page 9-624.
- *9.103 Instruction intrinsics* on page 9-625.
- *9.104 __breakpoint intrinsic* on page 9-626.
- *9.105 __cdp intrinsic* on page 9-627.
- *9.106 __clrex intrinsic* on page 9-628.
- *9.107 __clz intrinsic* on page 9-629.
- *9.108 __current_pc intrinsic* on page 9-630.
- *9.109 __current_sp intrinsic* on page 9-631.
- *9.110 __disable_fiq intrinsic* on page 9-632.
- *9.111 __disable_irq intrinsic* on page 9-633.
- *9.112 __dmb intrinsic* on page 9-635.
- *9.113 __dsb intrinsic* on page 9-636.
- *9.114 __enable_fiq intrinsic* on page 9-637.
- *9.115 __enable_irq intrinsic* on page 9-638.
- *9.116 __fabs intrinsic* on page 9-639.
- *9.117 __fabsf intrinsic* on page 9-640.
- *9.118 __force_loads intrinsic* on page 9-641.
- *9.119 __force_stores intrinsic* on page 9-642.
- *9.120 __isb intrinsic* on page 9-643.
- *9.121 __ldrex intrinsic* on page 9-644.
- *9.122 __ldrexd intrinsic* on page 9-646.
- *9.123 __ldrt intrinsic* on page 9-647.
- *9.124 __memory_changed intrinsic* on page 9-648.
- *9.125 __nop intrinsic* on page 9-649.
- *9.126 __pld intrinsic* on page 9-651.
- *9.127 __pli intrinsic* on page 9-652.
- *9.128 __promise intrinsic* on page 9-653.
- *9.129 __qadd intrinsic* on page 9-654.
- *9.130 __qdbl intrinsic* on page 9-655.
- *9.131 __qsub intrinsic* on page 9-656.
- *9.132 __rbit intrinsic* on page 9-657.
- *9.133 __rev intrinsic* on page 9-658.
- *9.134 __return_address intrinsic* on page 9-659.

- *9.135 __ror intrinsic* on page 9-660.
- *9.136 __schedule_barrier intrinsic* on page 9-661.
- *9.137 __semihost intrinsic* on page 9-662.
- *9.138 __sev intrinsic* on page 9-664.
- *9.139 __sqrt intrinsic* on page 9-665.
- *9.140 __sqrtf intrinsic* on page 9-666.
- *9.141 __ssat intrinsic* on page 9-667.
- *9.142 __strex intrinsic* on page 9-668.
- *9.143 __strexd intrinsic* on page 9-670.
- *9.144 __strt intrinsic* on page 9-672.
- *9.145 __swp intrinsic* on page 9-673.
- *9.146 __usat intrinsic* on page 9-674.
- *9.147 __wfe intrinsic* on page 9-675.
- *9.148 __wfi intrinsic* on page 9-676.
- *9.149 __yield intrinsic* on page 9-677.
- *9.150 ARMv6 SIMD intrinsics* on page 9-678.
- *9.151 ETSI basic operations* on page 9-679.
- *9.152 C55x intrinsics* on page 9-681.
- *9.153 VFP status intrinsic* on page 9-682.
- *9.154 __vfp_status intrinsic* on page 9-683.
- *9.155 Fused Multiply Add (FMA) intrinsics* on page 9-684.
- *9.156 Named register variables* on page 9-685.
- *9.157 GNU built-in functions* on page 9-689.
- *9.158 Predefined macros* on page 9-697.
- *9.159 Built-in function name variables* on page 9-703.

9.1 Keywords and operators

This topic lists the function keywords and operators that the compiler `armcc` supports.

The following table 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 9-1 Keyword extensions that the ARM compiler supports

Keywords		
`__align`	`__int64`	`__svc`
`__ALIGNOF__`	`__INTADDR__`	`__svc_indirect`
`__asm`	`__irq`	`__svc_indirect_r7`
`__declspec`	`__packed`	`__value_in_regs`
`__forceinline`	`__pure`	`__weak`
`__global_reg`	`__softfp`	`__writeonly`
`__inline`	`__smc`	

9.2 __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.

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.

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`**.

Restrictions

Because `__align` is a storage class modifier, it cannot be used on:

- Types, including **`typedef`**s 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.

Example

```
__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.
```

Related references

9.63 __attribute__((aligned)) variable attribute on page 9-582.

7.111 --min_array_alignment=opt on page 7-390.

9.3 __ALIGNOF__

The `__ALIGNOF__` keyword returns the alignment requirement for a specified type, or for the type of a specified object.

Syntax

`__ALIGNOF__(`*`type`*`)`

`__ALIGNOF__(`*`expr`*`)`

Where:

`type`
is a type

`expr`
is an lvalue.

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.

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
```

Related references

7.111 --min_array_alignment=opt on page 7-390.

9.4 __alignof__ on page 9-518.

9.4 __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.

Syntax

`__alignof__(`*`type`*`)`

`__alignof__(`*`expr`*`)`

Where:

`type`
is a type

`expr`
is an lvalue.

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.

Example

```
int Alignment_0(void)
            {
            return __alignof__(int);
            }
```

Related references

9.3 __ALIGNOF__ on page 9-517.

9.5 __asm

This keyword passes information from the compiler to the ARM assembler `armasm`.

The precise action of this keyword depends on its usage.

Usage

Embedded assembly

The `__asm` keyword can declare or define an embedded assembly function. For example:

```
__asm void my_strcpy(const char *src, char *dst);
```

Inline assembly

The `__asm` keyword can incorporate inline assembly into a function. For example:

```
int qadd(int i, int j)
{
    int res;
    __asm
    {
        QADD   res, i, j
    }
    return res;
}
```

Assembly labels

The `__asm` keyword can specify an assembly label for a C symbol. For example:

```
int count __asm__("count_v1"); // export count_v1, not count
```

Named register variables

The `__asm` keyword can declare a named register variable. For example:

```
register int foo __asm("r0");
```

Related concepts

6.26 Embedded assembler support in the compiler on page 6-243.

6.1 Compiler support for inline assembly language on page 6-216.

Related references

9.156 Named register variables on page 9-685.

8.36 Assembler labels on page 8-501.

9.6 __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 the function regardless of its characteristics.

In some circumstances the compiler may choose to ignore the `__forceinline` keyword and not inline a function. For example:

- A recursive function is never inlined into itself.
- Functions making use of `alloca()` are never inlined.

`__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))`.

———————————

Example

```
__forceinline static int max(int x, int y)
{
    return x > y ? x : y; // always inline if possible
}
```

Related references

7.65 --forceinline on page 7-339.

9.30 __attribute__((always_inline)) function attribute on page 9-549.

9.7 __global_reg

The `__global_reg` storage class specifier causes the compiler to reserve a register for a specific global variable.

Syntax

`__global_reg(`*`n`*`)` *`type varName`*

Where:

`n`
: Is an integer between one and eight.

`type`
: Is one of the following types:
  - Any integer type, except **`long long`**.
  - Any char type.
  - Any pointer type.

`varName`
: Is the name of a global variable.

Usage

`__global_reg` assigns a global variable to a specific register. It prevents the compiler from generating code that otherwise uses the register, in the same way as the `--global_reg` command-line option. The register number specified must be in the range 1-8. This corresponds to a register in the range r4 to r11.

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, ARM recommends 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.

Examples

This example declares a global variable `x` and reserves `r5` for it:

```
__global_reg(2) int x; // r5 is reserved for x
```

This example produces an error because global registers must be specified in all declarations of the same variable:

```
int x;
__global_reg(1) int x; // error
```

In C, `__global_reg` variables cannot be initialized at definition. This example produces an error in C, but not in C++:

```
__global_reg(1) int x=1; // error in C, OK in C++
```

Related references

7.72 --global_reg=reg_name[,reg_name,...] on page 7-349.

9.8 __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.

Example

```
__inline int f(int x)
{
    return x*5+1;
}
int g(int x, int y)
{
    return f(x) + f(y);
}
```

Related concepts

4.20 Inline functions on page 4-131.

Related references

7.65 --forceinline on page 7-339.

7.86 --inline, --no_inline on page 7-363.

9.6 __forceinline on page 9-520.

9.30 __attribute__((always_inline)) function attribute on page 9-549.

9.9 __int64

The `__int64` keyword is a synonym for the keyword sequence **`long long`**.

`__int64` is accepted even when using `--strict`.

Related references

8.12 long long on page 8-477.
7.156 --strict, --no_strict on page 7-439.

9.10 __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.

————————————

Syntax

`__INTADDR(`*`expr`*`)`

Where:

`expr`

is an integral constant expression.

Return value

`__INTADDR__(`*`expr`*`)` returns an integer constant equivalent to *`expr`*.

Related concepts

6.29 Restrictions on embedded assembly language functions in C and C++ code on page 6-246.

9.11 __irq

The `__irq` keyword enables a C or C++ function to be used as an exception handler.

`__irq` is a function qualifier. It affects the type of the function.

Usage

The `__irq` keyword causes the compiler to generate a function in a manner that makes it suitable for use as an exception handler. This means that the compiler makes the function:

- Preserve all processor registers, not only those required to be preserved by the AAPCS. Floating-point registers are not preserved.
- Return using an instruction that is architecturally defined as causing an exception return.

Restrictions

No arguments or return values can be used with `__irq` functions. `__irq` functions are incompatible with `--apcs /rwpi`.

Note

In ARMv6-M and ARMv7-M the architectural exception handling mechanism preserves all processor registers, and a standard function return can cause an exception return. Therefore, 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`.

Related references

7.160 --thumb on page 7-444.

7.7 --arm on page 7-277.

9.99 #pragma thumb on page 9-620.

9.76 #pragma arm on page 9-595.

Related information

ARM, Thumb, and ThumbEE instruction sets.

9.12 __packed

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.

Usage

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.

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.

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.

Errors

Taking the address of a field in a `__packed` structure or a `__packed`-qualified field yields a `__packed`-qualified pointer. The compiler produces 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.

Examples

This example shows that a pointer can point to a packed type.

```
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' */
```

This example 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.

```
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
}
```

Related concepts

4.35 The __packed qualifier and unaligned data access in C and C++ code on page 4-147.

4.40 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 4-152.

Related references

9.58 __attribute__((packed)) type attribute on page 9-577.

9.66 __attribute__((packed)) variable attribute on page 9-585.

9.95 #pragma pack(n) on page 9-615.

10.4 Structures, unions, enumerations, and bitfields in ARM C and C++ on page 10-710.

9.13 __pure

The `__pure` keyword asserts that a function declaration is pure.

Usage

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))`.

————————————

Pure functions are candidates for common subexpression elimination.

Default

By default, functions are assumed to be impure.

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.

Example

```
int factr(int n) __pure
{
    int f = 1;
    while (n > 0)
        f *= n--;
    return f;
}
```

Related concepts

4.17 Functions that return the same result when called with the same arguments on page 4-128.

4.19 Recommendation of postfix syntax when qualifying functions with ARM function modifiers on page 4-130.

Related references

4.18 Comparison of pure and impure functions on page 4-129.

9.31 __attribute__((const)) function attribute on page 9-550.

9.14 __smc

The `__smc` keyword declares an SMC (*Secure Monitor Call*) function.

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.

Usage

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.

Restrictions

The `SMC` instruction is available for selected ARM architecture-based processors, if they have the Security Extensions.

The compiler generates an error if you compile source code containing the `__smc` keyword for an architecture that does not support the `SMC` instruction.

Example

```
__smc(5) void mycall(void); /* declare a name by which SMC #5 can be called */
...
mycall();                   /* invoke the function */
```

Related references

7.29 --cpu=name compiler option on page 7-302.

Related information

SMC.

9.15 __softfp

The `__softfp` keyword asserts that a function uses software floating-point linkage. It is implicitly added to functions when softfp linkage is used.

`__softfp` is a function qualifier. It affects the type of the function.

Note

This keyword has the `#pragma` equivalent `#pragma __softfp_linkage`.

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.

Related concepts

4.49 Compiler support for floating-point computations and linkage on page 4-165.

Related references

7.69 --fpu=name compiler option on page 7-344.
9.98 #pragma softfp_linkage, #pragma no_softfp_linkage on page 9-619.
9.45 __attribute__((pcs("calling_convention"))) function attribute on page 9-564.

9.16 __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.

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.

Usage

This causes function invocations to be compiled inline as an AAPCS-compliant operation that behaves similarly to a normal call to a function.

You can use the `__value_in_regs` qualifier 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.

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.

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
```

Related references

9.17 __svc_indirect on page 9-533.
9.18 __svc_indirect_r7 on page 9-534.
7.29 --cpu=name compiler option on page 7-302.
9.19 __value_in_regs on page 9-535.

Related information

SVC.

9.17 __svc_indirect

You can use `__svc_indirect` to implement indirect SVCs.

Syntax

`__svc_indirect(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 `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.

Usage

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.

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.

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`.

Related references

9.16 __svc on page 9-532.
9.18 __svc_indirect_r7 on page 9-534.
7.29 --cpu=name compiler option on page 7-302.
9.19 __value_in_regs on page 9-535.

Related information

SVC.

9.18 __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.

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.

Usage

You can use this feature to implement indirect SVCs.

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`.

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.

Related references

9.16 __svc on page 9-532.
9.17 __svc_indirect on page 9-533.
7.29 --cpu=name compiler option on page 7-302.
9.19 __value_in_regs on page 9-535.

Related information

SVC.

9.19 __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.

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.

Usage

Declaring a function `__value_in_regs` can be useful when calling functions that return more than one result.

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.

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.

Example

```
typedef struct int64_struct
{
    unsigned int lo;
    unsigned int hi;
} int64_struct;
__value_in_regs extern
    int64_struct mul64(unsigned a, unsigned b);
```

Related concepts

4.16 Returning structures from functions through registers on page 4-127.

9.20 __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.

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.

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.

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 /rwpi
__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
```

Related information

--muldefweak, --no_muldefweak linker option.

9.21 __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.

Example

```
void foo(__writeonly int *ptr)
{
    *ptr = 0;                          // allowed
    printf("ptr value = %d\n", *ptr); // error
}
```

9.22 __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;
```

The following table summarizes the available `__declspec` attributes. `__declspec` attributes are storage class modifiers. They do not affect the type of a function or variable.

Table 9-2 __declspec attributes that the compiler supports, and their equivalents

__declspec attribute	non __declspec equivalent
`__declspec(noinline)`	`__attribute__((noinline))`
`__declspec(noreturn)`	`__attribute__((noreturn))`[a]
`__declspec(nothrow)`	-
`__declspec(notshared)`	-
`__declspec(thread)`	-

[a] A GNU compiler extension that the ARM compiler supports.

9.23 __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))`.

Example

```
/* 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);
```

Related references

9.38 __attribute__((noinline)) function attribute on page 9-557.

9.65 __attribute__((noinline)) constant variable attribute on page 9-584.

9.89 #pragma inline, #pragma no_inline on page 9-609.

9.24 __declspec(noreturn)

Informs the compiler that the function does not return. The compiler can then perform optimizations by removing code that is never reached.

Note

This attribute has the GNU-style equivalent `__attribute__((noreturn))`.

If the function reaches an explicit or implicit return, `__declspec(noreturn)` is ignored and the compiler generates a warning:

```
Warning:  #1461-D: function declared with "noreturn" does return
```

Usage

Use this attribute to reduce the cost of calling a function that never returns, such as `exit()`.

Best practice is to always terminate non-returning functions with `while(1);`.

Example

```
__declspec(noreturn) void overflow(void); // called on overflow

int negate(int x)
{
    if (x == 0x80000000) overflow();
    return -x;
}

void overflow(void)
{
    __asm {
        SVC 0x123; // hypothetical exception-throwing system service
    }
    while (1);
}
```

Related references

9.42 __attribute__((noreturn)) function attribute on page 9-561.

9.25 __declspec(nothrow)

The `__declspec(nothrow)` attribute asserts that a call to a function never results in a C++ exception being propagated from the callee 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 an exception. However, there are some restrictions on the unwinding tables produced for the C library functions that might throw an exception in a C++ context, for example, `bsearch` and `qsort`.

Note

This `__declspec` attribute has the function attribute equivalent `__attribute__((nothrow))`.

Usage

If the compiler knows that a function can never throw an exception, it might be able to generate smaller exception-handling tables for callers of that function.

Restrictions

If a call to a function results in a C++ exception being propagated from the callee into the caller, the behavior is undefined.

This modifier is ignored when not compiling with exceptions enabled.

Example

```
struct S
{
    ~S();
};
__declspec(nothrow) extern void f(void);
void g(void)
{
    S s;
    f();
}
```

Related references

7.64 --force_new_nothrow, --no_force_new_nothrow on page 7-338.
10.5 Using the ::operator new function in ARM C++ on page 10-715.
9.44 __attribute__((nothrow)) function attribute on page 9-563.

9.26 __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.

Example

```
struct __declspec(notshared) X
{
    virtual int f();
};                          // do not export this
int X::f()
{
    return 1;
}
struct Y : X
{
    virtual int g();
};                          // do export this
int Y::g()
{
    return 1;
}
```

9.27 __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)`.

————————————

Restrictions

File-scope thread-local variables cannot be dynamically initialized.

Example

```
__declspec(thread) int i;
__thread int j;          // same as __decspec(thread) int j;
```

9.28 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__));
```

The following table summarizes the available function attributes.

Table 9-3 Function attributes that the compiler supports, and their equivalents

Function attribute	Non-attribute equivalent
`__attribute__((alias))`	-
`__attribute__((always_inline))`	`__forceinline`
`__attribute__((const))`	`__pure`
`__attribute__((constructor[(`*priority*`)]))`	-
`__attribute__((deprecated))`	-
`__attribute__((destructor[(`*priority*`)]))`	-
`__attribute__((format_arg(`*string-index*`)))`	-
`__attribute__((malloc))`	-
`__attribute__((noinline))`	`__declspec(noinline)`
`__attribute__((nomerge))`	-
`__attribute__((nonnull))`	-
`__attribute__((noreturn))`	`__declspec(noreturn))`
`__attribute__((notailcall))`	-
`__attribute__((nothrow))`	`__declspec(nothrow))`
`__attribute__((pcs("`*calling_convention*`")))`	-
`__attribute__((pure))`	-
`__attribute__((section("`*name*`")))`	-
`__attribute__((unused))`	-
`__attribute__((used))`	-
`__attribute__((visibility("`*visibility_type*`")))`	-
`__attribute__((weak))`	`__weak`
`__attribute__((weakref("target")))`	-

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))`.

9.29 __attribute__((alias)) function attribute

This function attribute enables you to specify multiple aliases for a function.

Aliases must be defined in the same translation unit as the original function.

Note

You cannot specify aliases in block scope. The compiler ignores aliasing attributes attached to local function definitions and treats the function definition as a normal local definition.

In the output object file, the compiler replaces alias calls with a call to the original function name, and emits the alias alongside the original name. For example:

```
static int oldname(int x, int y) {
     return x + y;
}
static int newname(int x, int y) __attribute__((alias("oldname")));
int caller(int x, int y) {
     return oldname(x,y) + newname(x,y);
}
```

This code compiles to:

```
AREA ||.text||, CODE, READONLY, ALIGN=2
newname                  ; Alternate entry point
oldname PROC
     MOV      r2,r0
     ADD      r0,r2,r1
     BX       lr
     ENDP
caller PROC
     PUSH     {r4,r5,lr}
     MOV      r3,r0
     MOV      r4,r1
     MOV      r1,r4
     MOV      r0,r3
     BL       oldname
     MOV      r5,r0
     MOV      r1,r4
     MOV      r0,r3
     BL       oldname
     ADD      r0,r0,r5
     POP      {r4,r5,pc}
     ENDP
```

Note

This function attribute is a GNU compiler extension that the ARM compiler supports.

Note

Variables names might also be aliased using the corresponding variable attribute `__attribute__((alias))`.

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.

Example

```
#include <stdio.h>
void foo(void)
{
    printf("%s\n", __FUNCTION__);
}
void bar(void) __attribute__((alias("foo")));
void gazonk(void)
{
    bar(); // calls foo
}
```

Related references

9.61 __attribute__((alias)) variable attribute on page 9-580.

9.30 __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.

In some circumstances the compiler may choose to ignore the `__attribute__((always_inline))` attribute and not inline a function. For example:

- A recursive function is never inlined into itself.
- Functions making use of `alloca()` are never inlined.

Note

This function attribute is a GNU compiler extension that the ARM compiler supports. It has the keyword equivalent `__forceinline`.

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
}
```

Related references

7.65 --forceinline on page 7-339.

9.6 __forceinline on page 9-520.

9.31 __attribute__((const)) function attribute

The `const` function attribute specifies that a function examines only its arguments, and has no effect except for the return value. That is, the function does not read or modify any 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.

This is a much stricter class than `__attribute__((pure))` because functions are not permitted to read global memory.

Note

This function attribute is a GNU compiler extension that the ARM compiler supports. It has the keyword equivalent `__pure`.

Example

```
#include <stdio.h>

// __attribute__((const)) functions do not read or modify any global memory
int my_double(int b) __attribute__((const));
int my_double(int b) {
  return b*2;
}

int main(void) {
    int i;
    int result;
    for (i = 0; i < 10; i++)
    {
       result = my_double(i);
       printf (" i = %d ; result = %d \n", i, result);
    }
}
```

Related concepts

4.17 Functions that return the same result when called with the same arguments on page 4-128.

Related references

9.46 __attribute__((pure)) function attribute on page 9-565.

9.32 __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 that the ARM compiler supports.

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.

Usage

You can use this attribute for start-up or initialization code.

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(101)));
int my_constructor3(void) __attribute__((constructor(102)));
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;
}
```

Related references

--translate_g++.

--translate_gcc.

--translate_gld.

Related information

--init=symbol linker option.

9.33 __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.

In GNU mode, this attribute takes an optional string parameter to appear in the message, `__attribute__((deprecated("message")))`

Example

```
int Function_Attributes_deprecated_0(int b) __attribute__((deprecated));
```

9.34 __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 that the ARM compiler supports.

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.

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;
}
```

Related references

--translate_g++.

--translate_gcc.

--translate_gld.

Related information

--fini=symbol linker option.

9.35 __attribute__((format)) function attribute

This attribute causes the compiler to check that the supplied arguments are in the correct format for the specified function.

Syntax

`__attribute__((format(function, string-index, first-to-check)))`

Where `function` is a `printf`-style function, such as `printf()`, `scanf()`, `strftime()`, `gnu_printf()`, `gnu_scanf()`, `gnu_strftime()`, or `strfmon()`.

`string-index` specifies the index of the string argument in your function (starting from one).

`first-to-check` is the index of the first argument to check against the format string.

Example

```
#include <stdio.h>

extern char *myFormatText1 (const char *, ...);
extern char *myFormatText2 (const char *, ...) __attribute__((format(printf, 1, 2)));


int main(void) {
  int a, b;
  float c;

  a = 5;
  b = 6;
  c = 9.099999;

myFormatText1("Here are some integers: %d , %d\n", a, b); // No type checking. Types match.
myFormatText1("Here are some integers: %d , %d\n", a, c); // No type checking. Type
mismatch, but no warning.


myFormatText2("Here are some integers: %d , %d\n", a, b); // Type checking. Types match.
myFormatText2("Here are some integers: %d , %d\n", a, c); // Type checking. Warning: 181-D:
argument is incompatible...
}
```

`myFormatText1()` is a function that is given a string and two arguments to print. It has no format checking, so when it is passed a float argument and the function is expecting an integer, there is a silent type-mismatch.

`myFormatText2()` is identical to `myFormatText1()`, except it has `__attribute__((format()))`. When it receives an argument of an unexpected type, it raises a warning message.

9.36 __attribute__((format_arg(string-index))) function attribute

This attribute specifies that a function takes a format string as an argument. Format strings can contain typed placeholders that are intended to be passed to `printf`-style functions such as `printf()`, `scanf()`, `strftime()`, or `strfmon()`.

This attribute causes the compiler to perform placeholder type checking on the specified argument when the output of the function is used in calls to a `printf`-style function.

Note

This function attribute is a GNU compiler extension that the ARM compiler supports.

Syntax

`__attribute__((format_arg(`*`string-index`*`)))`

Where *`string-index`* specifies the argument that is the format string argument (starting from one).

Example

The following example declares two functions, `myFormatText1()` and `myFormatText2()`, that provide format strings to `printf()`.

The first function, `myFormatText1()`, does not specify the `format_arg` attribute. The compiler does not check the types of the `printf` arguments for consistency with the format string.

The second function, `myFormatText2()`, specifies the `format_arg` attribute. In the subsequent calls to `printf()`, the compiler checks that the types of the supplied arguments `a` and `b` are consistent with the format string argument to `myFormatText2()`. The compiler produces a warning when a `float` is provided where an `int` is expected.

```
#include <stdio.h>

// Function used by printf. No format type checking.
extern char *myFormatText1 (const char *);

// Function used by printf. Format type checking on argument 1.
extern char *myFormatText2 (const char *) __attribute__((format_arg(1)));


int main(void) {
  int a;
  float b;

  a = 5;
  b = 9.099999;

  printf(myFormatText1("Here is an integer: %d\n"), a); // No type checking. Types match
anyway.
  printf(myFormatText1("Here is an integer: %d\n"), b); // No type checking. Type mismatch,
but no warning

  printf(myFormatText2("Here is an integer: %d\n"), a); // Type checking. Types match.
  printf(myFormatText2("Here is an integer: %d\n"), b); // Type checking. Type mismatch
results in Warning:  #181-D
}


$ armcc format_arg_test.c -c
"format_arg_test.c", line 18: Warning:  #181-D: argument is incompatible with corresponding
format string conversion
     printf(myFormatText2("Here is an integer: %d\n"), b);
                                                       ^
format_arg_test.c: 1 warning, 0 errors
```

9.37 __attribute__((malloc)) function attribute

This function attribute indicates that the function can be treated like `malloc` and the compiler can perform the associated optimizations.

Example

```
void * foo(int b) __attribute__((malloc));
```

9.38 __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)`.In GNU mode, if this attribute is applied to a type instead of a function, the result is a warning rather than an error.

Example

```
int fn(void) __attribute__((noinline));
int fn(void)
{
    return 42;
}
```

Related references

9.65 __attribute__((noinline)) constant variable attribute on page 9-584.

9.89 #pragma inline, #pragma no_inline on page 9-609.

9.23 __declspec(noinline) on page 9-540.

9.39 __attribute__((no_instrument_function)) function attribute

Functions marked with this attribute are not profiled by `--gnu_instrument`.

Note

The `--gnu_instrument` option and this function attribute are deprecated from ARM Compiler 5.05 onwards.

Related references

7.75 --gnu_instrument, --no_gnu_instrument on page 7-352.

9.40 __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.

Related references

9.43 __attribute__((notailcall)) function attribute on page 9-562.

7.146 --retain=option on page 7-428.

9.41 __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.

Syntax

`__attribute__((nonnull[(`*`arg-index, ...`*`)]))`

Where `[(`*`arg-index, ...`*`)]` denotes an optional argument index list.

If no argument index list is specified, all pointer arguments are marked as nonnull.

Examples

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));
```

9.42 __attribute__((noreturn)) function attribute

Informs the compiler that the function does not return. The compiler can then perform optimizations by removing 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)`.

If the function reaches an explicit or implicit return, `__attribute__((noreturn))` is ignored and the compiler generates a warning:

```
Warning:  #1461-D: function declared with "noreturn" does return
```

Usage

Use this attribute to reduce the cost of calling a function that never returns, such as `exit()`.

Best practice is to always terminate non-returning functions with `while(1);`.

Example

```
void overflow(void) __attribute__((noreturn)); // called on overflow

int negate(int x)
{
    if (x == 0x80000000) overflow();
    return -x;
}

void overflow(void)
{
    __asm {
        SVC 0x123; // hypothetical exception-throwing system service
    }
    while (1);
}
```

Related references

9.24 __declspec(noreturn) on page 9-541.

9.43 __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.

Related references

9.40 __attribute__((nomerge)) function attribute on page 9-559.

7.146 --retain=option on page 7-428.

9.44 __attribute__((nothrow)) function attribute

This function attribute asserts that a call to a function never results in a C++ exception being propagated from the call into the caller.

——— **Note** ———

This function attribute is a GNU compiler extension that the ARM compiler supports. It has the `__declspec` equivalent `__declspec(nothrow)`.

———————

Example

```
struct S
{
    ~S();
};
extern void f(void) __attribute__((nothrow));
void g(void)
{
    S s;
    f();
}
```

9.45 __attribute__((pcs("calling_convention"))) function attribute

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 that the ARM compiler supports.

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.

Related concepts

4.49 Compiler support for floating-point computations and linkage on page 4-165.

Related references

9.15 __softfp on page 9-531.

9.46 __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))`.

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.

Related concepts

4.17 Functions that return the same result when called with the same arguments on page 4-128.

Related references

9.31 __attribute__((const)) function attribute on page 9-550.

9.47 __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.

Examples

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
```

Related references

9.77 #pragma arm section [section_type_list] on page 9-596.

9.48 __attribute__((sentinel)) function attribute

This function attribute generates a warning if the specified parameter in a function call is not NULL.

Syntax

`__attribute__ ((sentinel(p)))`

Where:

p

is an optional integer position argument. If this argument is supplied, the compiler checks the parameter at position *p* counting backwards from the end of the argument list.

By default, the compiler checks the parameter at position zero, the last parameter of the function call. That is, `__attribute__ ((sentinel))` is equivalent to `__attribute__ ((sentinel(0)))`

9.49 __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.

————————————

Example

```
static int Function_Attributes_unused_0(int b) __attribute__((unused));
```

9.50 __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.

Functions marked with `__attribute__((used))` are tagged in the object file to avoid removal by linker unused section removal.

Note

Static variables can also be marked as used using `__attribute__((used))`.

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
```

Related references

9.69 __attribute__((used)) variable attribute on page 9-588.

9.47 __attribute__((section("name"))) function attribute on page 9-566.

Related information

Elimination of unused sections.

9.51 __attribute__((visibility("visibility_type"))) function attribute

This function attribute affects the visibility of ELF symbols.

Note

This attribute is a GNU compiler extension that the ARM compiler supports.

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 *processor-specific Application Binary Interface* (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.

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.

Example

```
void __attribute__((visibility("internal"))) foo()
{
   ...
}
```

Related references

9.70 __attribute__((visibility("visibility_type"))) variable attribute on page 9-589.

--arm_linux.

--visibility_inlines_hidden.

--hide_all, --no_hide_all.

9.52 __attribute__((warn_unused_result))

In GNU-mode, warn if a function returns a result that is never used.

9.53 __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.

———————————

Example

```
extern int Function_Attributes_weak_0 (int b) __attribute__((weak));
```

Related references

9.71 __attribute__((weak)) variable attribute on page 9-590.

9.20 __weak on page 9-536.

9.54 __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.

Syntax

```
__attribute__((weakref("target")))
```

Where *`target`* is the target symbol.

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();
  ...
}
```

Restrictions

This attribute can only be used on functions with static linkage.

9.55 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__));
```

The following table summarizes the available type attributes.

Table 9-4 Type attributes that the compiler supports, and their equivalents

Type attribute	Non-attribute equivalent
`__attribute__((bitband))`	-
`__attribute__((aligned))`	`__align`
`__attribute__((packed))`	`__packed`[b]
`__attribute__((transparent_union))`	-

[b] The `__packed` qualifier does not affect type in GNU mode.

9.56 __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.

The following example illustrates the use of `__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 the following example, bit-banded access is generated for `bb.i`.

```
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.

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.

Related references

7.13 --bitband on page 7-283.

9.62 __attribute__((at(address))) variable attribute on page 9-581.

Related information

--rw_base=address (linker option).

9.57 __attribute__((aligned)) type attribute

The `aligned` type attribute specifies a minimum alignment for the type.

9.58 __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.

In non-GNU mode, this attribute is equivalent to `__packed`, and has stronger constraints than when used in GNU-mode.

GNU mode

To enable GNU-mode, use the `--gnu` option.

In GNU mode, `__attribute__((packed))` has the effect of `#pragma packed`. Taking the address of a field covered by `__attribute__((packed))` does not produce a `__packed` qualified pointer.

In non-GNU-mode, `__attribute__((packed))` has the effect of `__packed`. Taking the address of a field covered by `__attribute__((packed))` produces a `__packed` qualified pointer.

```
struct foobar {
  char x;
  short y[10] __attribute__((packed));
};

short get_y0(struct foobar *s){
    return *s->y;  // Unaligned-capable load
}

short *get_y(struct foobar *s){
    return s->y;   // Compile error unless --gnu is used.
}
```

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 produces 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`.

Related concepts

4.35 The __packed qualifier and unaligned data access in C and C++ code on page 4-147.

4.40 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 4-152.

Related references

7.73 --gnu on page 7-350.

9.66 __attribute__((packed)) variable attribute on page 9-585.

9.95 #pragma pack(n) on page 9-615.

9.12 __packed on page 9-527.

10.4 Structures, unions, enumerations, and bitfields in ARM C and C++ on page 10-710.

9.59 __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.

———————————

Mode

Supported in GNU mode only.

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 */
}
```

9.60 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__));
```

The following table summarizes the available variable attributes.

Table 9-5 Variable attributes that the compiler supports, and their equivalents

Variable attribute	Non-attribute equivalent
`__attribute__((alias))` on page 9-580	-
`__attribute__((at(address)))` on page 9-581	-
`__attribute__((aligned))` on page 9-582	-
`__attribute__((deprecated))` on page 9-583	-
`__attribute__((noinline))` on page 9-584	
`__attribute__((packed))` on page 9-585	-
`__attribute__((section("name")))` on page 9-586	-
`__attribute__((unused))` on page 9-587	-
`__attribute__((used))` on page 9-588	-
`__attribute__((visibility("visibility_type")))` on page 9-589	-
`__attribute__((weak))` on page 9-590	`__weak` on page 9-536
`__attribute__((weakref("target")))` on page 9-591	
`__attribute__((zero_init))` on page 9-592	-

9.61 __attribute__((alias)) variable attribute

This variable attribute enables you to specify multiple aliases for a variable.

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.

Usage

Aliases must be defined in the same translation unit as the original variable.

Note

You cannot specify aliases in block scope. The compiler ignores aliasing attributes attached to local variable definitions and treats the variable definition as a normal local definition.

In the output object file, the compiler replaces alias references with a reference to the original variable name, and emits the alias alongside the original name. For example:

```
int oldname = 1;
extern int newname __attribute__((alias("oldname")));
```

This code compiles to:

```
     LDR      r1,[r0,#0]  ; oldname
     ...
oldname
newname
     DCD      0x00000001
```

If the original variable is defined as **`static`** but the alias is defined as **`extern`**, then the compiler changes the original variable to be external.

Note

Function names might also be aliased using the corresponding function attribute `__attribute__((alias))`.

Example

```
#include <stdio.h>
int oldname = 1;
extern int newname __attribute__((alias("oldname"))); // declaration
void foo(void)
{
    printf("newname = %d\n", newname); // prints 1
}
```

Related references

9.29 __attribute__((alias)) function attribute on page 9-547.

9.62 __attribute__((at(address))) variable attribute

This variable attribute enables you to specify the absolute address of a variable.

Syntax

`__attribute__((at(`*`address`*`)))`

Where:

`address`

is the desired address of the variable.

Usage

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

GNU compilers do not support this variable attribute.

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.

Errors

The linker gives an error message if it is not possible to place a section at a specified address.

Example

```
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 */
```

Related information

Placement of __at sections at a specific address.

9.63 __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.

Example

```
/* 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));
```

Related references

9.2 __align on page 9-516.

9.64 __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.

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.

9.65 __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.

You can use this feature 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.

Example

```
__attribute__((noinline)) const int m = 1;
```

Related references

9.38 __attribute__((noinline)) function attribute on page 9-557.

9.89 #pragma inline, #pragma no_inline on page 9-609.

9.23 __declspec(noinline) on page 9-540.

9.66 __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.

Example

```
struct
{
    char a;
    int b __attribute__((packed));
} Variable_Attributes_packed_0;
```

Related concepts

4.35 The __packed qualifier and unaligned data access in C and C++ code on page 4-147.

4.40 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 4-152.

Related references

9.58 __attribute__((packed)) type attribute on page 9-577.

9.95 #pragma pack(n) on page 9-615.

9.12 __packed on page 9-527.

10.4 Structures, unions, enumerations, and bitfields in ARM C and C++ on page 10-710.

9.67 __attribute__((section("name"))) variable attribute

The `section` attribute specifies that a variable must be placed in a particular data section.

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.

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 that the ARM compiler supports.

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));
```

Related information

How to find where a symbol is placed when linking.

9.68 __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 to issue a warning if it is not used.

Note

This variable attribute is a GNU compiler extension that the ARM compiler supports.

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.

9.69 __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.

Usage

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))`.

You can use `__attribute__((used))` to build tables in the object.

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
```

Related references

9.50 __attribute__((used)) function attribute on page 9-569.

9.47 __attribute__((section("name"))) function attribute on page 9-566.

Related information

Elimination of unused sections.

9.70 __attribute__((visibility("visibility_type"))) variable attribute

This variable attribute affects the visibility of ELF symbols.

Note

This attribute is a GNU compiler extension that the ARM compiler supports.

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 *processor-specific Application Binary Interface* (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.

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.

Example

```
int i __attribute__((visibility("hidden")));
```

Related references

9.51 __attribute__((visibility("visibility_type"))) function attribute on page 9-570.
--arm_linux.
--visibility_inlines_hidden.
--hide_all, --no_hide_all.

9.71 __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.

Related references

9.53 __attribute__((weak)) function attribute on page 9-572.

9.20 __weak on page 9-536.

9.72 __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 that the ARM compiler supports.

Syntax

```
__attribute__((weakref("target")))
```

Where `target` is the target symbol.

Restrictions

This attribute can only be used on variables that are declared as static.

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;
  ...
}
```

9.73 __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.

Example

```
__attribute__((zero_init)) int x;                   /* in section ".bss" */
__attribute__((section("mybss"), zero_init)) int y;  /* in section "mybss" */
```

Related references

9.47 __attribute__((section("name"))) function attribute on page 9-566.

9.74 Pragmas

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`.

The following table summarizes the available pragmas.

Table 9-6 Pragmas that the compiler supports

Pragmas		
`#pragma anon_unions`, `#pragma no_anon_unions`	`#pragma hdrstop`	`#pragma pack(`*n*`)`
`#pragma arm`	`#pragma import` *symbol_name*	`#pragma pop`
`#pragma arm section [`*section_type_list*`]`	`#pragma import(__use_full_stdio)`	`#pragma push`
`#pragma diag_default` *tag*`[,`*tag*`,...]`	`#pragma import(__use_smaller_memcpy)`	`#pragma softfp_linkage`, `no_softfp_linkage`
`#pragma diag_error` *tag*`[,`*tag*`,...]`	`#pragma inline`, `#pragma no_inline`	`#pragma unroll [(`*n*`)]`
`#pragma diag_remark` *tag*`[,`*tag*`,...]`	`#pragma no_pch`	`#pragma unroll_completely`
`#pragma diag_suppress` *tag*`[,`*tag*`,...]`	`#pragma O`*num*	`#pragma thumb`
`#pragma diag_warning` *tag*`[,`*tag*`,...]`	`#pragma once`	`#pragma weak` *symbol*
`#pragma [no_]exceptions_unwind`	`#pragma Ospace`	`#pragma weak` *symbol1* `=` *symbol2*
`#pragma GCC system_header`	`#pragma Otime`	

9.75 #pragma anon_unions, #pragma no_anon_unions

These pragmas enable and disable support for anonymous structures and unions.

Default

The default is `#pragma no_anon_unions`.

Related references

8.35 Anonymous classes, structures and unions on page 8-500.
9.59 __attribute__((transparent_union)) type attribute on page 9-578.

9.76 #pragma arm

This pragma switches code generation to the ARM instruction set. It overrides the `--thumb` compiler option.

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.

Related references

9.96 #pragma pop on page 9-617.

9.97 #pragma push on page 9-618.

9.99 #pragma thumb on page 9-620.

7.7 --arm on page 7-277.

7.160 --thumb on page 7-444.

9.77 #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`.

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`.

Usage

Use `#pragma arm section [`*`section_type_list`*`]` to place functions and variables in separate named sections. You can then use the scatter-loading description file to locate these at a particular address in memory.

Restrictions

This option has no effect on:

- Inline functions and their local static variables if the `--no_ool_section_name` command-line option is specified.
- 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.

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
```

Related references

9.47 __attribute__((section("name"))) function attribute on page 9-566.

7.123 --ool_section_name, --no_ool_section_name on page 7-405.

Related information

Scatter-loading Features.

9.78 #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`.

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. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.

At least one diagnostic message number must be specified.

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.

Related concepts

5.3 Controlling compiler diagnostic messages with pragmas on page 5-209.

Related references

9.79 #pragma diag_error tag[,tag,...] on page 9-599.

9.80 #pragma diag_remark tag[,tag,...] on page 9-600.

9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.

9.82 #pragma diag_warning tag[, tag, ...] on page 9-602.

7.48 --diag_warning=tag[,tag,...] on page 7-322.

9.79 #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`.

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. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.

  At least one diagnostic message number must be specified.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

9.80 #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_error`, 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.

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. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

9.81 #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_error`, except that the compiler suppresses the diagnostic messages having the specified tags rather than setting them to have Error severity.

Syntax

`#pragma diag_suppress` *`tag`*`[,`*`tag`*`,...]`

Where:

`tag``[,`*`tag`*`,...]`
: is a comma-separated list of diagnostic message numbers specifying the messages to be suppressed. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.

Related references

7.15 --brief_diagnostics, --no_brief_diagnostics on page 7-286.
7.43 --diag_error=tag[,tag,...] on page 7-317.
7.44 --diag_remark=tag[,tag,...] on page 7-318.
7.45 --diag_style=arm|ide|gnu compiler option on page 7-319.
7.46 --diag_suppress=tag[,tag,...] on page 7-320.
7.47 --diag_suppress=optimizations on page 7-321.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.178 --wrap_diagnostics, --no_wrap_diagnostics on page 7-463.
7.49 --diag_warning=optimizations on page 7-323.
7.57 --errors=filename on page 7-331.
7.173 -W on page 7-458.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
7.143 --remarks on page 7-425.
Chapter 5 Compiler Diagnostic Messages on page 5-205.

9.82 #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_error`, except that the compiler sets the diagnostic messages having the specified tags to Warning severity rather than Error severity.

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. This is the four-digit number, *`nnnn`*, with the tool letter prefix, but without the letter suffix indicating the severity.

Related concepts

5.3 Controlling compiler diagnostic messages with pragmas on page 5-209.

Related references

9.78 #pragma diag_default tag[,tag,...] on page 9-598.
9.79 #pragma diag_error tag[,tag,...] on page 9-599.
9.80 #pragma diag_remark tag[,tag,...] on page 9-600.
9.81 #pragma diag_suppress tag[,tag,...] on page 9-601.
7.48 --diag_warning=tag[,tag,...] on page 7-322.

9.83 #pragma exceptions_unwind, #pragma no_exceptions_unwind

These pragmas enable and disable function unwinding.

Usage

The `--[no_]exceptions_unwind` command-line option sets the default behavior for whether unwind tables are generated for functions. `#pragma [no_]exceptions_unwind` overrides this behavior.

Default

The default is `#pragma exceptions_unwind`.

Related references

10.11 C++ exception handling in ARM C++ on page 10-722.

7.59 --exceptions_unwind, --no_exceptions_unwind on page 7-333.

7.58 --exceptions, --no_exceptions on page 7-332.

9.84 #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.

Related references

7.73 --gnu on page 7-350.

9.85 #pragma hdrstop

This pragma enables you to specify where the set of precompilation header files end.

Note

This pragma is deprecated.

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

This pragma must appear before the first token that does not belong to a preprocessing directive.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.23 Precompiled Header (PCH) file processing and the header stop point on page 3-91.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.

Related references

7.30 --create_pch=filename on page 7-304.
7.129 --pch on page 7-411.
7.130 --pch_dir=dir on page 7-412.
7.131 --pch_messages, --no_pch_messages on page 7-413.
7.132 --pch_verbose, --no_pch_verbose on page 7-414.
7.166 --use_pch=filename on page 7-451.
9.90 #pragma no_pch on page 9-610.

9.86 #pragma import symbol_name

This pragma generates an importing reference to `symbol_name`.

This is the same as the assembler directive:

```
IMPORT symbol_name
```

Syntax

`#pragma import symbol_name`

Where:

`symbol_name`
is a symbol to be imported.

Usage

You can use this pragma to select certain features of the C library, such as the heap implementation or real-time division. For example:

```
#pragma import(__use_realtime_division)
```

If a feature described in this book requires a symbol reference to be imported, the required symbol is specified.

Related information

IMPORT and EXTERN (assembler directives).
Using the C library with an application.

9.87 #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.

Related references

7.94 --library_type=lib on page 7-372.

Related information

About microlib.
Tailoring input/output functions in the C and C++ libraries.

9.88 #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.

———————————

Default

The default version of `memcpy()` used by microlib is a larger, but faster, word-by-word implementation using `LDR` and `STR`.

Related references

7.94 --library_type=lib on page 7-372.

Related information

The ARM C Micro-library.

9.89 #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))`.

Default

The default is `#pragma inline`.

Related references

7.86 --inline, --no_inline on page 7-363.

9.38 __attribute__((noinline)) function attribute on page 9-557.

9.65 __attribute__((noinline)) constant variable attribute on page 9-584.

9.23 __declspec(noinline) on page 9-540.

9.90 #pragma no_pch

This pragma suppresses *Precompiled Header* (PCH) processing.

Note

This pragma is deprecated.

Support for Precompiled Header (PCH) files is deprecated from ARM Compiler 5.05 onwards on all platforms. Note that ARM Compiler on Windows 8 never supported PCH files.

Related concepts

3.21 Precompiled Header (PCH) files on page 3-88.
3.24 Precompiled Header (PCH) file creation requirements on page 3-93.
3.28 Selectively applying Precompiled Header (PCH) file processing on page 3-98.
3.29 Suppressing Precompiled Header (PCH) file processing on page 3-99.

Related references

7.30 --create_pch=filename on page 7-304.
7.129 --pch on page 7-411.
7.130 --pch_dir=dir on page 7-412.
7.131 --pch_messages, --no_pch_messages on page 7-413.
7.132 --pch_verbose, --no_pch_verbose on page 7-414.
7.166 --use_pch=filename on page 7-451.
9.85 #pragma hdrstop on page 9-605.

9.91 #pragma Onum

This pragma changes the optimization level for all subsequent functions.

Syntax

`#pragma O`*`num`*

Where:

`num`

is the new optimization level.

The value of *`num`* is 0, 1, 2 or 3.

Usage

To assign a new optimization level to all subsequent functions, use `#pragma Onum`. For example, compiling with `armcc -O1`:

```
void function1(void){
    ...                  // Optimized at O1 (from armcc -O1)
}

#pragma O3
void function2(void){
    ...                  // Optimized at O3
}

void function3(void){
    ...                  // Optimized at O3
}
```

To assign a new optimization level to an individual function, use `#pragma Onum` together with `#pragma push` and `#pragma pop`. For example, compiling with `armcc -O1`:

```
void function1(void){
    ...                   // Optimized at O1 (from armcc -O1)
}
#pragma push
#pragma O3
void function2(void){
    ...                   // Optimized at O3
}
#pragma pop

void function3(void){
    ...                   // Optimized at O1 (from armcc -O1)
}
```

Restriction

The pragma must be placed outside a function.

Related references

9.93 #pragma Ospace on page 9-613.

9.94 #pragma Otime on page 9-614.

7.119 -Onum on page 7-399.

9.92 #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.

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.

9.93 #pragma Ospace

This pragma optimizes all subsequent functions for code size, performing optimizations to reduce image size at the expense of a possible increase in execution time.

Usage

To optimize all subsequent functions for code size, use `#pragma Ospace`. For example, when compiling with `armcc -Otime`:

```
void function1(void){
    ...                  // Optimized for execution time (from armcc -Otime)
}

#pragma Ospace
void function2(void){
    ...                  // Optimized for code size
}

void function3(void){
    ...                  // Optimized for code size
}
```

To optimize an individual function for code size, use `#pragma Ospace` together with `#pragma push` and `#pragma pop`. For example, when compiling with `armcc -Otime`:

```
void function1(void){
    ...                   // Optimized for execution time (from armcc -Otime)
}
#pragma push
#pragma Ospace
void function2(void){
    ...                   // Optimized for code size
}
#pragma pop

void function3(void){
    ...                   // Optimized for execution time (from armcc -Otime)
}
```

Restriction

The pragma must be placed outside a function.

Related references

7.124 -Ospace on page 7-406.
9.91 #pragma Onum on page 9-611.
9.94 #pragma Otime on page 9-614.
7.119 -Onum on page 7-399.

9.94 #pragma Otime

This pragma optimizes all subsequent functions for speed, performing optimizations to reduce execution time at the expense of a possible increase in image size.

Usage

To optimize all subsequent functions for execution time, use `#pragma Otime`. For example, when compiling with `armcc -Ospace` (the default):

```
void function1(void){
    ...                  // Optimized for code size (from armcc -Ospace)
}

#pragma Otime
void function2(void){
    ...                  // Optimized for execution time
}

void function3(void){
    ...                  // Optimized for execution time
}
```

To optimize an individual function for execution time, use `#pragma Otime` together with `#pragma push` and `#pragma pop`. For example, when compiling with `armcc -Ospace` (the default):

```
void function1(void){
    ...                   // Optimized for code size (from armcc -Ospace)
}
#pragma push
#pragma Otime
void function2(void){
    ...                   // Optimized for execution time
}
#pragma pop

void function3(void){
    ...                   // Optimized for code size (from armcc -Ospace)
}
```

Restriction

The pragma must be placed outside a function.

Related references

9.91 #pragma Onum on page 9-611.
9.93 #pragma Ospace on page 9-613.
7.119 -Onum on page 7-399.
7.125 -Otime on page 7-407.

9.95 #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.

Syntax

`#pragma pack(n)`

Where:

n

is the alignment in bytes, valid alignment values being `1`, `2`, `4` and `8`.

Default

The default is `#pragma pack(8)`.

Errors

Taking the address of a field in a `#pragma packed` **`struct`** does not yield a `__packed` pointer, so the compiler does 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.

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:

0 a	1 padding	2	3
4 b	5 b	6 b	7 b

Figure 9-1 Nonpacked structure S

The layout of `SP` is:

0 a	1 x	2 b	3 b
4 b	5 b		

Figure 9-2 Packed structure SP

Note

In this layout, `x` denotes one byte of padding.

`SP` is a 6-byte structure. There is no padding after `b`.

Related concepts

4.35 The __packed qualifier and unaligned data access in C and C++ code on page 4-147.

4.40 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 4-152.

Related references

9.66 __attribute__((packed)) variable attribute on page 9-585.

9.58 __attribute__((packed)) type attribute on page 9-577.

9.12 __packed on page 9-527.

10.4 Structures, unions, enumerations, and bitfields in ARM C and C++ on page 10-710.

9.96 #pragma pop

This pragma restores the previously saved pragma state.

Related concepts

5.3 Controlling compiler diagnostic messages with pragmas on page 5-209.

Related references

9.97 #pragma push on page 9-618.

9.97 #pragma push

This pragma saves the current pragma state.

Related concepts

5.3 Controlling compiler diagnostic messages with pragmas on page 5-209.

Related references

9.96 #pragma pop on page 9-617.

9.98 #pragma softfp_linkage, #pragma no_softfp_linkage

These pragmas control software floating-point linkage.

`#pragma softfp_linkage` adds an implicit `__softfp` qualifier to all subsequent function declarations and definitions.

`#pragma no_softfp_linkage` removes any implicit `__softfp` qualifiers from all subsequent function declarations and definitions. Explicit `__softfp` qualifiers are still respected.

When using command-line options that enable software floating-point linkage, implicit `__softfp` qualifiers are added to each function.

Default

The default floating-point linkage type depends on the target FPU architecture.

Related concepts

4.51 Compiler options for floating-point linkage and computations on page 4-167.

Related references

9.15 __softfp on page 9-531.

9.99 #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.

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.

```
#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();
}
```

Related references

7.7 --arm on page 7-277.

7.160 --thumb on page 7-444.

9.76 #pragma arm on page 9-595.

9.96 #pragma pop on page 9-617.

9.97 #pragma push on page 9-618.

9.100 #pragma unroll [(n)]

This pragma instructs the compiler to unroll a loop by *n* iterations.

Syntax

`#pragma unroll`

`#pragma unroll (n)`

Where:

n

is an optional value indicating the number of iterations to unroll.

Default

If you do not specify a value for *n*, the compiler assumes `#pragma unroll (4)`.

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.

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.

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*.

Related concepts

4.7 Loop unrolling in C code on page 4-115.

Related references

9.101 #pragma unroll_completely on page 9-623.
7.48 --diag_warning=tag[,tag,...] on page 7-322.
7.119 -Onum on page 7-399.
7.125 -Otime on page 7-407.

9.101 #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.

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.

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.

Related concepts

4.7 Loop unrolling in C code on page 4-115.

Related references

9.100 #pragma unroll [(n)] on page 9-621.

7.48 --diag_warning=tag[,tag,...] on page 7-322.

7.119 -Onum on page 7-399.

7.125 -Otime on page 7-407.

9.102 #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.

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)
{
    ...
}
```

Related references

9.61 __attribute__((alias)) variable attribute on page 9-580.

9.29 __attribute__((alias)) function attribute on page 9-547.

9.53 __attribute__((weak)) function attribute on page 9-572.

9.71 __attribute__((weak)) variable attribute on page 9-590.

9.103 Instruction intrinsics

This topic describes instruction intrinsics for realizing ARM machine language instructions from C or C++ code.

The following table summarizes the available intrinsics.

Table 9-7 Instruction intrinsics that the ARM compiler supports

Instruction intrinsics			
`__breakpoint`	`__cdp`	`__clrex`	`__clz`
`__current_pc`	`__current_sp`	`__disable_fiq`	`__disable_irq`
`__dmb`	`__dsb`	`__enable_fiq`	`__enable_irq`
`__fabs`	`__fabsf`	`__force_loads`	`__force_stores`
`__isb`	`__ldrex`	`__ldrexd`	`__ldrt`
`__memory_changed`	`__nop`	`__pld`	`__pldw`
`__pli`	`__promise`	`__qadd`	`__qdbl`
`__qsub`	`__rbit`	`__return_address`	`__rev`
`__ror`	`__schedule_barrier`	`__semihost`	`__sev`
`__sqrt`	`__sqrtf`	`__ssat`	`__strex`
`__strexd`	`__strt`	`__swp`	`__usat`
`__wfe`	`__wfi`	`__yield`	

9.104 __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.

Syntax

`void __breakpoint(int val)`

Where:

val

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.

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, depending on the source language:

- In C code: `Warning: #223-D: function "__breakpoint" declared implicitly`.
- In C++ code: `Error: #20: identifier "__breakpoint" is undefined`.

The undefined instruction trap is taken if a `BKPT` instruction is executed on an architecture that does not support it.

Example

```
void func(void)
{
    ...
    __breakpoint(0xF02C);
    ...
}
```

Related information

BKPT.

9.105 __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.

Syntax

`__cdp(unsigned int coproc, unsigned int ops, unsigned int regs)`

Where:

`coproc`

Identifies the coprocessor the instruction is for.

`coproc` must be an integer in the range 0 to 15.

`ops`

Is an encoding of the two opcodes for the `CDP` or `CDP2` instruction, `(opcode1<<4) | opcode2`, where:

- The first opcode, *`opcode1`*, occupies the 4-bit coprocessor-specific opcode field in the instruction.
- The second opcode, *`opcode2`*, occupies the 3-bit coprocessor-specific opcode field in the instruction.

Add `0x100` to *`ops`* to generate a `CDP2` instruction.

`regs`

Is an encoding of the coprocessor registers, `(CRd<<8) | (CRn<<4) | CRm`, where *`CRd`*, *`CRn`* and *`CRm`* are the coprocessor registers for the `CDP` or `CDP2` instruction.

Usage

The use of these instructions depends on the coprocessor. See your coprocessor documentation for more information.

Example

```
void copro_example()
{
    const unsigned int ops = 0xA3; // opcode1 = 0xA, opcode2 = 0x3
    const unsigned int regs = 0xCDE; // CRd = 0xC (12), CRn = 0xD (13), CRm = 0xE (14)
    __cdp(4,ops,regs); // coprocessor number 4
    // This intrinsic produces the instruction CDP p4,#0xa,c12,c13,c14,#3
}
```

Related information

CDP and CDP2.

9.106 __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.

Syntax

```
void __clrex(void)
```

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, depending on the source language:

- In C code: `Warning: #223-D: function "__clrex" declared implicitly`.
- In C++ code: `Error: #20: identifier "__clrex" is undefined`.

Related information

CLREX.

9.107 __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.

Syntax

`unsigned char __clz(unsigned int *val*)`

Where:

`val`
: is an **`unsigned int`**.

Return value

The `__clz` intrinsic returns the number of leading zeros in *`val`*.

Related references

9.157 GNU built-in functions on page 9-689.

Related information

CLZ.

9.108 __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.

Syntax

```
unsigned int __current_pc(void)
```

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.

Related references

9.109 __current_sp intrinsic on page 9-631.

9.134 __return_address intrinsic on page 9-659.

9.109 __current_sp intrinsic

This intrinsic returns the value of the stack pointer at the current point in your program.

Syntax

```
unsigned int __current_sp(void)
```

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.

Related references

9.108 __current_pc intrinsic on page 9-630.
9.134 __return_address intrinsic on page 9-659.
9.157 GNU built-in functions on page 9-689.

9.110 __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.

Syntax

`int __disable_fiq(void);`

`void __disable_fiq(void);`

Usage

`int __disable_fiq(void);` disables fast interrupts and returns the value the FIQ interrupt mask has in the `PSR` before disabling interrupts.

`void __disable_fiq(void);` disables fast interrupts.

Return value

`int __disable_fiq(void);` returns the value the FIQ interrupt mask has in the `PSR` before disabling FIQ interrupts.

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 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`.

Example

```
void foo(void)
{
    int was_masked = __disable_fiq();
    /* ... */
    if (!was_masked)
        __enable_fiq();
}
```

Related references

9.114 __enable_fiq intrinsic on page 9-637.

9.111 __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).

Syntax

`int __disable_irq(void);`

`void __disable_irq(void);`

Usage

`int __disable_irq(void);` disables interrupts and returns the value the IRQ interrupt mask has in the `PSR` before disabling interrupts.

`void __disable_irq(void);` disables interrupts.

Return value

`int __disable_irq(void);` returns the value the IRQ interrupt mask has in the `PSR` before disabling IRQ interrupts.

Example

```
void foo(void)
{
    int was_masked = __disable_irq();
    /* ... */
    if (!was_masked)
        __enable_irq();
}
```

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 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 shows 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:  f00000080 ....  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`.

Related references

9.115 __enable_irq intrinsic on page 9-638.

9.112 __dmb intrinsic

This intrinsic inserts a `DMB` or equivalent instruction into the instruction stream generated by the compiler.

The `DMB` instruction ensures the observed ordering of memory accesses.

If the target does not support the `DMB` instruction, the compiler treats this intrinsic as an optimization barrier.

Syntax

```
void __dmb(unsigned int val)
```

Where *`val`* is a numeric argument indicating the scope and access type of the barrier. See *ARM C Language Extensions* for more information.

Related references

9.124 __memory_changed intrinsic on page 9-648.
9.136 __schedule_barrier intrinsic on page 9-661.
9.119 __force_stores intrinsic on page 9-642.
9.118 __force_loads intrinsic on page 9-641.
9.113 __dsb intrinsic on page 9-636.
9.120 __isb intrinsic on page 9-643.

Related information

ARM C Language Extensions.

9.113 __dsb intrinsic

This intrinsic inserts a DSB or equivalent instruction into the instruction stream generated by the compiler.

The DSB instruction ensures the completion of memory accesses.

If the target does not support the DSB instruction, the compiler treats this intrinsic as an optimization barrier.

Syntax

```
void __dsb(unsigned int val)
```

Where *val* is a numeric argument indicating the scope and access type of the barrier. See *ARM C Language Extensions* for more information.

Related references

9.124 __memory_changed intrinsic on page 9-648.
9.136 __schedule_barrier intrinsic on page 9-661.
9.119 __force_stores intrinsic on page 9-642.
9.112 __dmb intrinsic on page 9-635.
9.118 __force_loads intrinsic on page 9-641.
9.120 __isb intrinsic on page 9-643.

Related information

ARM C Language Extensions.

9.114 __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.

Syntax

```
void __enable_fiq(void)
```

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`.

Related references

9.110 __disable_fiq intrinsic on page 9-632.

9.115 __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).

Syntax

```
void __enable_irq(void)
```

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`.

Related references

9.111 __disable_irq intrinsic on page 9-633.

9.116 __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.

Syntax

```
double __fabs(double val)
```

Where:

`val`
is a double-precision floating-point value.

Return value

The `__fabs` intrinsic returns the absolute value of *`val`* as a **`double`**.

Related references

9.117 __fabsf intrinsic on page 9-640.

Related information

VABS (floating-point).

9.117 __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.

Syntax

```
float __fabs(float val)
```

9.118 __force_loads 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 reloaded from memory.

This intrinsic also acts as a scheduling barrier.

Syntax

```
void __force_loads(void)
```

Related references

9.124 __memory_changed intrinsic on page 9-648.

9.136 __schedule_barrier intrinsic on page 9-661.

9.119 __force_stores intrinsic on page 9-642.

9.112 __dmb intrinsic on page 9-635.

9.113 __dsb intrinsic on page 9-636.

9.120 __isb intrinsic on page 9-643.

9.119 __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.

Syntax

```
void __force_stores(void)
```

Related references

9.124 __memory_changed intrinsic on page 9-648.
9.136 __schedule_barrier intrinsic on page 9-661.
9.118 __force_loads intrinsic on page 9-641.
9.112 __dmb intrinsic on page 9-635.
9.113 __dsb intrinsic on page 9-636.
9.120 __isb intrinsic on page 9-643.

9.120 __isb intrinsic

This intrinsic inserts an `ISB` or equivalent instruction into the instruction stream generated by the compiler.

The `ISB` instruction flushes the processor pipeline fetch buffers, so that subsequent instructions are fetched from cache or memory.

If the target does not support the `ISB` instruction, the compiler treats this intrinsic as an optimization barrier.

Syntax

```
void __isb(unsigned int val)
```

Where *`val`* is a numeric argument indicating the scope and access type of the barrier. The only supported value for the `__isb` intrinsic is `15`. See *ARM C Language Extensions* for more information.

Related references

9.124 __memory_changed intrinsic on page 9-648.
9.136 __schedule_barrier intrinsic on page 9-661.
9.119 __force_stores intrinsic on page 9-642.
9.112 __dmb intrinsic on page 9-635.
9.113 __dsb intrinsic on page 9-636.
9.118 __force_loads intrinsic on page 9-641.

Related information

ARM C Language Extensions.

9.121 __ldrex intrinsic

The `__ldrex` intrinsic lets you load data from memory in your C or C++ code using an `LDREX[size]` instruction.

`size` in `LDREX[size]` is `B` for byte loads or `H` for halfword loads. If no size is specified, word loads are performed.

Note

This intrinsic is deprecated.

Note

The compiler does not guarantee to preserve the state of the exclusive monitor. It might generate load and store instructions between the `LDREX` instruction generated for the `__ldrex` intrinsic and the `STREX` instruction generated for the `__strex` intrinsic. Because memory accesses can clear the exclusive monitor, code using the `__ldrex` and `__strex` intrinsics can have unexpected behavior. Where `LDREX` and `STREX` instructions are needed, ARM recommends using embedded assembly.

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 9-8 Access widths that the __ldrex intrinsic supports

Instruction	Size of data loaded	Pointer type
`LDREXB`	byte	**`unsigned char *`**
`LDREXB`	byte	**`signed char *`**
`LDREXH`	halfword	**`unsigned short *`**
`LDREXH`	halfword	**`signed short *`**
`LDREX`	word	**`int *`**

Return value

The `__ldrex` intrinsic returns the data loaded from the memory address pointed to by *`ptr`*.

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, depending on the source language:

- In C code: `Warning: #223-D: function "__ldrex" declared implicitly`.
- In C++ code: `Error: #20: identifier "__ldrex" is undefined`.

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.

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
```

Related references

9.122 __ldrexd intrinsic on page 9-646.

9.142 __strex intrinsic on page 9-668.

9.143 __strexd intrinsic on page 9-670.

Related information

LDREX.

9.122 __ldrexd intrinsic

The `__ldrexd` intrinsic lets you load doubleword data from memory in your C or C++ code using an `LDREXD` instruction.

Note

This intrinsic is deprecated.

Note

The compiler does not guarantee to preserve the state of the exclusive monitor. It might generate load and store instructions between the `LDREXD` instruction generated for the `__ldrexd` intrinsic and the `STREXD` instruction generated for the `__strexd` intrinsic. Because memory accesses can clear the exclusive monitor, code using the `__ldrexd` and `__strexd` intrinsics can have unexpected behavior. Where `LDREXD` and `STREXD` instructions are needed, ARM recommends using embedded assembly.

Syntax

`unsigned long long __ldrexd(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 9-9 Access widths that the __ldrex intrinsic supports

Instruction	Size of data loaded	Pointer type
`LDREXD`	long long	**`long long *`**

Return value

The `__ldrexd` intrinsic returns the data loaded from the memory address pointed to by *`ptr`*.

Errors

The compiler does not recognize the `__ldrexd` 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, depending on the source language:

- In C code: `Warning: #223-D: function "__ldrexd" declared implicitly`.
- In C++ code: `Error: #20: identifier "__ldrexd" is undefined`.

The `__ldrexd` intrinsic only supports access to doubleword data. The compiler generates an error if you specify an access width that is not supported.

Related references

9.121 __ldrex intrinsic on page 9-644.
9.142 __strex intrinsic on page 9-668.
9.143 __strexd intrinsic on page 9-670.

Related information

LDREX.

9.123 __ldrt intrinsic

The `__ldrt` intrinsic lets you load data from memory in your C or C++ code using an `LDR{size}T` instruction.

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 9-10 Access widths that the __ldrt intrinsic supports

Instruction[c]	Size of data loaded	Pointer type
`LDRSBT`	byte	**`signed char *`**
`LDRBT`	byte	**`unsigned char *`**
`LDRSHT`	halfword	**`signed short *`**
`LDRHT`	halfword	**`unsigned short *`**
`LDRT`	word	**`int *`**

Return value

The `__ldrt` intrinsic returns the data loaded from the memory address pointed to by *`ptr`*.

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, depending on the source language:

- In C code: `Warning: #223-D: function "__ldrt" declared implicitly`.
- In C++ code: `Error: #20: identifier "__ldrt" is undefined`.

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.

Example

```
int foo(void)
{
    int loc = 0xff;
    return __ldrt((const volatile int *)loc);
}
```

Compiling this code with the default options produces:

```
||foo|| PROC
    MOV      r1,#0xff
    LDRT     r0,[r1],#0
    BX       lr
    ENDP
```

Related references

7.160 --thumb on page 7-444.

Related information

LDR, unprivileged.

[c] If the target instruction set does not have the listed instruction, the compiler generates a sequence of instructions with equivalent behavior instead.

9.124 __memory_changed intrinsic

This intrinsic causes the compiler to behave as if all C objects had their values both read and written at that point in time.

The compiler ensures that the stored value of each C object is correct at that point in time and treats the stored value as unknown afterwards.

Syntax

```
void __memory_changed(void)
```

Related references

9.119 __force_stores intrinsic on page 9-642.
9.136 __schedule_barrier intrinsic on page 9-661.
9.118 __force_loads intrinsic on page 9-641.
9.112 __dmb intrinsic on page 9-635.
9.113 __dsb intrinsic on page 9-636.
9.120 __isb intrinsic on page 9-643.

9.125 __nop intrinsic

This intrinsic inserts a NOP instruction or an equivalent code sequence into the instruction stream.

Syntax

`void __nop(void)`

Usage

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.

Examples

In the following example, the compiler ensures that the read from the volatile variable x is enclosed between two NOP instructions.

```
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-M3`, for example, then the compiler can schedule the read of the non-volatile variable z to be before the read of variable x.

In the following example, the compiler ensures that the write to variable z is enclosed between two NOP instructions.

```
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 the following example, because `pure_func` has no side effects, the compiler can move the call to it to outside of the loop. Still, the compiler ensures that the call to `func` is enclosed between two `NOP` instructions.

```
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`.

Related references

9.13 __pure on page 9-529.
9.136 __schedule_barrier intrinsic on page 9-661.
3.4 Generic intrinsics on page 3-67.
9.138 __sev intrinsic on page 9-664.
9.147 __wfe intrinsic on page 9-675.
9.148 __wfi intrinsic on page 9-676.
9.149 __yield intrinsic on page 9-677.

Related information

NOP.

9.126 __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.

Syntax

`void __pld(...)`

Where:

`...`
: denotes any number of pointer or integer arguments specifying addresses of memory to prefetch.

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.

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;
}
```

Related references

9.127 __pli intrinsic on page 9-652.

Related information

PLD and PLI.

9.127 __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.

Syntax

```
void __pli(...)
```

Where:

`...`
: denotes any number of pointer or integer arguments specifying addresses of instructions to prefetch.

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.

Related references

9.126 __pld intrinsic on page 9-651.

Related information

PLD and PLI.

9.128 __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.

Syntax

```
void __promise(expr)
```

Where *expr* is an expression that evaluates to nonzero.

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)`.

Related concepts

Indicating loop iteration counts to the compiler with __promise(expr).

9.129 __qadd intrinsic

This intrinsic inserts a `QADD` instruction into the instruction stream generated by the compiler. It enables you to obtain the result of a saturating add of two integers from within your C or C++ code.

Note

The compiler might optimize your code when it detects an opportunity to do so, using equivalent instructions from the same family to produce fewer instructions.

Syntax

`int __qadd(int` *`val1`*`, int` *`val2`*`)`

Where:

`val1`
is the first summand of the saturating add operation

`val2`
is the second summand of the saturating add operation.

Return value

The `__qadd` intrinsic returns the saturating add of *`val1`* and *`val2`*.

Errors

The compiler does not recognize the `__qadd` intrinsic when compiling for a target that does not support the `QADD` instruction. The compiler generates either a warning or an error in this case, depending on the source language:

- In C code: `Warning: #223-D: function "__qadd" declared implicitly`.
- In C++ code: `Error: #20: identifier "__qadd" is undefined`.

Related references

9.130 __qdbl intrinsic on page 9-655.

9.131 __qsub intrinsic on page 9-656.

Related information

QADD.

9.130 __qdbl intrinsic

This intrinsic inserts instructions equivalent to the saturating addition 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.

Syntax

`int __qdbl(int val)`

Where:

`val`
: is the data value to be doubled.

Return value

The `__qdbl` intrinsic returns the saturating add of *`val`* with itself, or equivalently, `__qadd(val, val)`.

Errors

The compiler does not recognize the `__qdbl` intrinsic when compiling for a target that does not support the `QADD` instruction. The compiler generates either a warning or an error in this case, depending on the source language:

- In C code: `Warning: #223-D: function "__qdbl" declared implicitly`.
- In C++ code: `Error: #20: identifier "__qdbl" is undefined`.

Related references

9.129 __qadd intrinsic on page 9-654.

9.131 __qsub intrinsic on page 9-656.

9.131 __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.

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.

Syntax

`int __qsub(int val1, int val2)`

Where:

`val1`
is the minuend of the saturating subtraction operation

`val2`
is the subtrahend of the saturating subtraction operation.

Return value

The `__qsub` intrinsic returns the saturating subtraction of *`val1`* and *`val2`*.

Errors

The compiler does not recognize the `__qsub` intrinsic when compiling for a target that does not support the `QSUB` instruction. The compiler generates either a warning or an error in this case, depending on the source language:

- In C code: `Warning: #223-D: function "__qsub" declared implicitly`.
- In C++ code: `Error: #20: identifier "__qsub" is undefined`.

Related references

9.129 __qadd intrinsic on page 9-654.

9.130 __qdbl intrinsic on page 9-655.

Related information

QSUB.

9.132 __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.

Syntax

```
unsigned int __rbit(unsigned int val)
```

Where:

`val`
: is the data value whose bit order is to be reversed.

Return value

The `__rbit` intrinsic returns the value obtained from `val` by reversing its bit order.

Errors

The compiler does not recognize the `__rbit` intrinsic when compiling for a target that does not support the `RBIT` instruction. The compiler generates either a warning or an error in this case, depending on the source language:

- In C code: `Warning: #223-D: function "__rbit" declared implicitly`.
- In C++ code: `Error: #20: identifier "__rbit" is undefined`.

Related information

RBIT.

9.133 __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.

Syntax

```
unsigned int __rev(unsigned int val)
```

Where:

`val`
: is an **`unsigned int`**.

Return value

The `__rev` intrinsic returns the value obtained from `val` by reversing its byte order.

Related information

REV.

9.134 __return_address intrinsic

This intrinsic returns the return address of the current function.

Syntax

```
unsigned int __return_address(void)
```

Return value

The `__return_address` intrinsic returns the value of the link register that is used in returning from the current function.

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()`.

Related references

9.108 __current_pc intrinsic on page 9-630.

9.109 __current_sp intrinsic on page 9-631.

9.157 GNU built-in functions on page 9-689.

9.135 __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.

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.

Return value

The `__ror` intrinsic returns the value of `val` rotated right by `shift` number of places.

Related information

ROR.

9.136 __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.

Syntax

```
void __schedule_barrier(void)
```

Related references

9.119 __force_stores intrinsic on page 9-642.
9.124 __memory_changed intrinsic on page 9-648.
9.125 __nop intrinsic on page 9-649.
9.118 __force_loads intrinsic on page 9-641.
9.112 __dmb intrinsic on page 9-635.
9.113 __dsb intrinsic on page 9-636.
9.120 __isb intrinsic on page 9-643.

9.137 __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.

Syntax

`int __semihost(int val, const void *ptr)`

Where:

`val`
Is the request code for the semihosting request.

`ptr`
Is a pointer to an argument/result block.

Return value

The results of semihosting calls are passed either as an explicit return value or as a pointer to a data block.

Usage

Use this intrinsic from C or C++ to generate the appropriate semihosting call for your target and instruction set:

`0x123456`
In ARM state for all architectures.

`0xAB`
In Thumb state, excluding M-profile architectures. This behavior is not guaranteed on *all* debug targets from ARM or from third parties.

`0xAB`
For M-profile architectures (Thumb only).

Restrictions

ARM processors earlier than 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.

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
```

Related concepts

11.2 The semihosting interface on page 11-730.

Related references

7.28 --cpu=list on page 7-301.

7.160 --thumb on page 7-444.

9.16 __svc on page 9-532.

Related information

BKPT.

SVC.

9.138 __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.

Syntax

```
void __sev(void)
```

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, depending on the source language:

- In C code: `Warning: #223-D: function "__sev" declared implicitly`.
- In C++ code: `Error: #20: identifier "__sev" is undefined`.

Related references

9.125 __nop intrinsic on page 9-649.

9.147 __wfe intrinsic on page 9-675.

9.148 __wfi intrinsic on page 9-676.

9.149 __yield intrinsic on page 9-677.

Related information

NOP.

SEV.

9.139 __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.

Syntax

`double __sqrt(double *val*)`

Where:

`val`
is a double-precision floating-point value.

Return value

The `__sqrt` intrinsic returns the square root of *`val`* as a **`double`**.

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, depending on the source language:

- In C code: `Warning: #223-D: function "__sqrt" declared implicitly`.
- In C++ code: `Error: #20: identifier "__sqrt" is undefined`.

Related references

9.140 __sqrtf intrinsic on page 9-666.

9.140 __sqrtf intrinsic

This intrinsic is a single-precision version of the `__sqrt` 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.

Related references

9.139 __sqrt intrinsic on page 9-665.

9.141 __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.

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.

Return value

The `__ssat` intrinsic returns *`val`* saturated to the signed range $-2^{sat-1} \leq x \leq 2^{sat-1} - 1$.

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, depending on the source language:

- In C code: `Warning: #223-D: function "__ssat" declared implicitly.`
- In C++ code: `Error: #20: identifier "__ssat" is undefined.`

Related references

9.146 __usat intrinsic on page 9-674.

Related information

SSAT.

9.142 __strex intrinsic

The `__strex` intrinsic lets you use an `STREX[size]` instruction in your C or C++ code to store data to memory.

——— **Note** ———

This intrinsic is deprecated.

———

——— **Note** ———

The compiler does not guarantee to preserve the state of the exclusive monitor. It might generate load and store instructions between the `LDREX` instruction generated for the `__ldrex` intrinsic and the `STREX` instruction generated for the `__strex` intrinsic. Because memory accesses can clear the exclusive monitor, code using the `__ldrex` and `__strex` intrinsics can have unexpected behavior. Where `LDREX` and `STREX` instructions are needed, ARM recommends using embedded assembly.

———

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 9-11 Access widths that the __strex intrinsic supports

Instruction	Size of data stored	Pointer type
`STREXB`	byte	**`char`** `*`
`STREXH`	halfword	**`short`** `*`
`STREX`	word	**`int`** `*`

Return value

The `__strex` intrinsic returns:

`0`
: if the `STREX` instruction succeeds

`1`
: if the `STREX` instruction is locked out.

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, depending on the source language:

- In C code: `Warning: #223-D: function "__strex" declared implicitly`.
- In C++ code: `Error: #20: identifier "__strex" is undefined`.

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.

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]
    CMP      r1,#0
    MOVEQ    r0,#1
    MOVNE    r0,#0
    BX       lr
    ENDP
```

Related references

9.121 __ldrex intrinsic on page 9-644.

9.122 __ldrexd intrinsic on page 9-646.

9.143 __strexd intrinsic on page 9-670.

Related information

STREX.

9.143 __strexd intrinsic

The `__strexd` intrinsic lets you use an `STREXD` instruction in your C or C++ code to perform an exclusive doubleword data store to memory.

Note

This intrinsic is deprecated.

Note

The compiler does not guarantee to preserve the state of the exclusive monitor. It might generate load and store instructions between the `LDREXD` instruction generated for the `__ldrexd` intrinsic and the `STREXD` instruction generated for the `__strexd` intrinsic. Because memory accesses can clear the exclusive monitor, code using the `__ldrexd` and `__strexd` intrinsics can have unexpected behavior. Where `LDREXD` and `STREXD` instructions are needed, ARM recommends using embedded assembly.

Syntax

```
int __strexd(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 9-12 Access widths that the __strexd intrinsic supports

Instruction	Size of data stored	Pointer type
`STREXD`	long long	**`long long *`**

Return value

The `__strexd` intrinsic returns:

`0`
: if the `STREXD` instruction succeeds

`1`
: if the `STREXD` instruction is locked out.

Errors

The compiler does not recognize the `__strexd` 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, depending on the source language:

- In C code: `Warning: #223-D: function "__strexd" declared implicitly`.
- In C++ code: `Error: #20: identifier "__strexd" is undefined`.

The `__strexd` intrinsic only supports access to doubleword data. The compiler generates an error if you specify an access width that is not supported.

Related references

9.121 __ldrex intrinsic on page 9-644.
9.122 __ldrexd intrinsic on page 9-646.
9.142 __strex intrinsic on page 9-668.

Related information

STREX.

9.144 __strt intrinsic

The `__strt` intrinsic lets you store data to memory in your C or C++ code using an `STR{size}T` instruction.

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 9-13 Access widths that the __strt intrinsic supports

Instruction	Size of data stored	Pointer type
STRBT	byte	**char** *
STRHT	halfword	**short** *
STRT	word	**int** *

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, depending on the source language:

- In C code: `Warning: #223-D: function "__strt" declared implicitly`.
- In C++ code: `Error: #20: identifier "__strt" is undefined`.

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. The unused most-significant bits of `val` are ignored when signed data is stored.

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
```

Related references

7.160 --thumb on page 7-444.

Related information

STR, unprivileged.

9.145 __swp intrinsic

The __swp intrinsic lets you use a SWP{size} instruction to swap data between registers and memory from your C or C++ code.

The SWP{size} instruction reads a value into a processor register and writes a value to a memory location as an atomic operation.

Note

The use of SWP and SWPB is deprecated in ARMv6 and above.

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 9-14 Access widths that the __swp intrinsic supports

Instruction	Size of data	Pointer type
SWPB	byte	**char ***
SWP	word	**int ***

Return value

The __swp intrinsic returns the data value in the memory address pointed to by *ptr* immediately before the SWP instruction overwrites it with *val*.

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
```

Related information

SWP and SWPB.

9.146 __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.

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.

`usat` must be in the range 0 to 31.

Return value

The `__usat` intrinsic returns *`val`* saturated to the unsigned range $0 \leq x \leq 2^{sat} - 1$.

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, depending on the source language:

- In C code: `Warning: #223-D: function "__usat" declared implicitly`.
- In C++ code: `Error: #20: identifier "__usat" is undefined`.

Related references

9.141 __ssat intrinsic on page 9-667.

Related information

USAT.

9.147 __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.

Syntax

```
void __wfe(void)
```

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, depending on the source language:

- In C code: `Warning: #223-D: function "__wfe" declared implicitly`.
- In C++ code: `Error: #20: identifier "__wfe" is undefined`.

Related references

9.138 __sev intrinsic on page 9-664.

9.125 __nop intrinsic on page 9-649.

9.148 __wfi intrinsic on page 9-676.

9.149 __yield intrinsic on page 9-677.

Related information

NOP.

WFE.

9.148 __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.

Syntax

```
void __wfi(void)
```

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, depending on the source language:

- In C code: `Warning: #223-D: function "__wfi" declared implicitly`.
- In C++ code: `Error: #20: identifier "__wfi" is undefined`.

Related references

9.138 __sev intrinsic on page 9-664.

9.125 __nop intrinsic on page 9-649.

9.147 __wfe intrinsic on page 9-675.

9.149 __yield intrinsic on page 9-677.

Related information

NOP.

WFI.

9.149 __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.

Syntax

```
void __yield(void)
```

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, depending on the source language:

- In C code: `Warning: #223-D: function "__yield" declared implicitly`.
- In C++ code: `Error: #20: identifier "__yield" is undefined`.

Related references

9.138 __sev intrinsic on page 9-664.

9.125 __nop intrinsic on page 9-649.

9.147 __wfe intrinsic on page 9-675.

9.148 __wfi intrinsic on page 9-676.

Related information

NOP.

YIELD.

9.150 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 ARMv6 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.

Related references

9.156 Named register variables on page 9-685.

Chapter 12 ARMv6 SIMD Instruction Intrinsics on page 12-758.

Related information

SEL.

VFP Programming.

ARM registers.

9.151 ETSI basic operations

The compilation tools support the original ETSI family of basic operations through intrinsics.

The original ETSI family of basic operations are 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 the following table.

Table 9-15 ETSI basic operations that the ARM compilation tools support

Intrinsics				
abs_s	L_add_c	L_mult	L_sub_c	norm_l
add	L_deposit_h	L_negate	mac_r	round
div_s	L_deposit_l	L_sat	msu_r	saturate
extract_h	L_mac	L_shl	mult	shl
extract_l	L_macNs	L_shr	mult_r	shr
L_abs	L_msu	L_shr_r	negate	shr_r
L_add	L_msuNs	L_sub	norm_s	sub

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 the following table.

Table 9-16 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.

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
}
```

Related concepts

3.9 Compiler support for European Telecommunications Standards Institute (ETSI) basic operations on page 3-72.

Related information

ETSI Recommendation G.191: Software tools for speech and audio coding standardization.

ETSI Recommendation G.729: Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP).

ETSI Recommendation G723.1 : Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s.

9.152 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 the following table.

Table 9-17 TI C55x intrinsics that the compilation tools support

Intrinsics			
`_a_lsadd`	`_a_sadd`	`_a_smac`	`_a_smacr`
`_a_smas`	`_a_smasr`	`_abss`	`_count`
`_divs`	`_labss`	`_lmax`	`_lmin`
`_lmpy`	`_lmpysu`	`_lmpyu`	`_lnorm`
`_lsadd`	`_lsat`	`_lshl`	`_shrs`
`_lsmpy`	`_lsmpyi`	`_lsmpyr`	`_lsmpysu`
`_lsmpysui`	`_lsmpyu`	`_lsmpyui`	`_lsneg`
`_lsshl`	`_lssub`	`_max`	`_min`
`_norm`	`_rnd`	`_round`	`_roundn`
`_sadd`	`_shl`	`_shrs`	`_smac`
`_smaci`	`_smacr`	`_smacsu`	`_smacsui`
`_smas`	`_smasi`	`_smasr`	`_smassu`
`_smassui`	`_smpy`	`_sneg`	`_sround`
`_sroundn`	`_sshl`	`_ssub`	-

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`.

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
}
```

9.153 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.

Related references

9.154 __vfp_status intrinsic on page 9-683.

9.156 Named register variables on page 9-685.

9.154 __vfp_status intrinsic

This intrinsic reads or modifies the FPSCR.

Syntax

```
unsigned int __vfp_status(unsigned int mask, unsigned int flags);
```

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 the following table. The intrinsic returns the modified value of FPSCR if `mask` and `flags` are not both 0.

Table 9-18 Modifying the FPSCR flags

`mask` bit	`flags` 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>`.

Errors

The compiler generates an error if you attempt to use this intrinsic when compiling for a target that does not have VFP.

Related concepts

4.78 <fenv.h> floating-point environment access in C99 on page 4-200.

Related information

VFP system registers.

9.155 Fused Multiply Add (FMA) intrinsics

These intrinsics perform the calculation `result = a × b + c`, incurring only a single rounding step.

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-M4 processor.
- If compiling for the Cortex-M4 processor, only `fmaf()` is available.

9.156 Named register variables

The compiler enables you to access registers of an ARM architecture-based processor or coprocessor using named register variables.

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 the following table.

Table 9-19 Named registers available on ARM architecture-based processors

Register	Character string for `__asm`	Processors
`APSR`	`"apsr"`	All processors
`CPSR`	`"cpsr"`	All processors, apart from Cortex-M series processors.
`BASEPRI`	`"basepri"`	ARMv7-M processors
`BASEPRI_MAX`	`"basepri_max"`	ARMv7-M processors
`CONTROL`	`"control"`	ARMv6-M and ARMv7-M processors
`DSP`	`"dsp"`	ARMv6-M and ARMv7-M processors
`EAPSR`	`"eapsr"`	ARMv6-M and ARMv7-M processors
`EPSR`	`"epsr"`	ARMv6-M and ARMv7-M processors
`FAULTMASK`	`"faultmask"`	ARMv7-M processors
`IAPSR`	`"iapsr"`	ARMv6-M and ARMv7-M processors
`IEPSR`	`"iepsr"`	ARMv6-M and ARMv7-M processors
`IPSR`	`"ipsr"`	ARMv6-M and ARMv7-M processors
`MSP`	`"msp"`	ARMv6-M and ARMv7-M processors
`PRIMASK`	`"primask"`	ARMv6-M and ARMv7-M processors
`PSP`	`"psp"`	ARMv6-M and ARMv7-M processors
`PSR`	`"psr"`	ARMv6-M and ARMv7-M processors
`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"`	ARMv6-M and ARMv7-M processors

On targets with floating-point hardware, the registers listed in the following table are also available for use with named register variables.

Table 9-20 Named registers available on targets with floating-point hardware

Register	Character string for __asm
`FPSID`	`"fpsid"`
`FPSCR`	`"fpscr"`
`FPEXC`	`"fpexc"`
`FPINST`	`"fpinst"`
`FPINST2`	`"fpinst2"`

Table 9-20 Named registers available on targets with floating-point hardware (continued)

Register	Character string for __asm
FPSR	"fpsr"
MVFR0	"mvfr0"
MVFR1	"mvfr1"

Note

Some registers are not available on some architectures.

Usage

You must declare core registers as global rather than local named register variables. Your program might still compile if you declare them locally, but you risk unexpected runtime behavior if you do. There is no restriction on the scope of named register variables for other registers.

Note

A global named register variable is global to the source file in which it is declared, not global to the program. It has no effect on other files, unless you use multifile compilation or you declare it in a header file.

Restrictions

Declaring a core register as a named register variable means that register is not available to the compiler for other operations. If you declare too many named register variables, code size increases significantly. In some cases, your program might not compile, for example if there are insufficient registers available to compute a particular expression.

Register usage defined by the *Procedure Call Standard for the ARM Architecture* (AAPCS) is not affected by declaring named register variables. For example, `r0` is always used to return result values from functions even if it is declared as a named register variable.

Named register variables are a compiler-only feature. With the exception of `r12`, tools such as linkers do not change register usage in object files. The AAPCS reserves `r12` as the inter-procedural scratch register. You must not declare `r12` as a named register variable if you require its value to be preserved across function calls.

Examples

In the following example, `apsr` is declared as a named register variable for the `"apsr"` register:

```
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 the following example, `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:

```
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 the following example, `cp15_control` is declared as a register variable for accessing a coprocessor register. This example enables the MMU using `CP15`:

```
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
```

The following example for Cortex-M3 declares `_msp`, `_control` and `_psp` as named register variables to set up stack pointers:

```
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;        // set up 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
```

Related concepts

3.12 Compiler support for accessing registers using named register variables on page 3-76.

Related information

Procedure Call Standard for the ARM Architecture.

9.157 GNU built-in functions

These functions provide compatibility with GNU library header files. The functions are described in the GNU documentation.

See *GCC, the GNU Compiler Collection* for more information.

Nonstandard functions

- `__builtin_alloca()`
- `__builtin_bcmp()`
- `__builtin_exit()`
- `__builtin_gamma()`
- `__builtin_gammaf()`
- `__builtin_gammal()`
- `__builtin_index()`
- `__builtin___memcpy_chk()`
- `__builtin___memmove_chk()`
- `__builtin_mempcpy()`
- `__builtin___mempcpy_chk()`
- `__builtin___memset_chk()`
- `__builtin_object_size()`
- `__builtin_rindex()`
- `__builtin___snprintf_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()`

C99 functions

- `__builtin_exit()`
- `__builtin_acoshf()`
- `__builtin_acoshl()`
- `__builtin_acosh()`
- `__builtin_asinhf(`
- `__builtin_asinhl()`
- `__builtin_asinh()`
- `__builtin_atanhf()`
- `__builtin_atanhl()`
- `__builtin_atanh()`
- `__builtin_cabsf()`
- `__builtin_cabsl()`
- `__builtin_cabs()`
- `__builtin_cacosf()`
- `__builtin_cacoshf()`
- `__builtin_cacoshl()`
- `__builtin_cacosh()`
- `__builtin_cacosl()`
- `__builtin_cacos()`
- `__builtin_cargf()`

- `__builtin_cargl()`
- `__builtin_carg()`
- `__builtin_casinf()`
- `__builtin_casinhf()`
- `__builtin_casinhl()`
- `__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_crealf()`
- `__builtin_creall()`
- `__builtin_creal()`
- `__builtin_csinf()`
- `__builtin_csinhf()`
- `__builtin_csinhl()`
- `__builtin_csinh()`
- `__builtin_csinl()`
- `__builtin_csin()`
- `__builtin_csqrtf()`
- `__builtin_csqrtl()`
- `__builtin_csqrt()`

- `__builtin_ctanf()`
- `__builtin_ctanhf()`
- `__builtin_ctanhl()`
- `__builtin_ctanh()`
- `__builtin_ctanl()`
- `__builtin_ctan()`
- `__builtin_erfcf()`
- `__builtin_erfcl()`
- `__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_snprintf()`
- `__builtin_tgammaf()`
- `__builtin_tgammal()`
- `__builtin_tgamma()`
- `__builtin_truncf()`
- `__builtin_truncl()`
- `__builtin_trunc()`
- `__builtin_vfscanf()`
- `__builtin_vscanf()`
- `__builtin_vsnprintf()`
- `__builtin_vsscanf()`

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_ceill()`
- `__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_frexpf()`
- `__builtin_frexpl()`
- `__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()`
- `__builtin_sqrtl()`
- `__builtin_tanf()`
- `__builtin_tanhf()`
- `__builtin_tanhl()`
- `__builtin_tanl()`

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()

C90 functions

- __builtin_abort()
- __builtin_abs()
- __builtin_acos()
- __builtin_asin()
- __builtin_atan2()
- __builtin_atan()
- __builtin_calloc()
- __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_snprintf()

- `__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`.

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()`

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()`

Other built-in functions

- `__builtin_choose_expr()`
- `__builtin_clz()`
- `__builtin_types_compatible_p()`
- `__builtin_constant_p()`
- `__builtin_ctz()`
- `__builtin_ctzl()`
- `__builtin_ctzll()`
- `__builtin_expect()`
- `__builtin_ffs()`
- `__builtin_ffsl()`
- `__builtin_ffsll()`
- `__builtin_frame_address()`
- `__builtin_offsetof()`
- `__builtin_prefetch()`
- `__builtin_return_address()`
- `__builtin_popcount()`
- `__builtin_signbit()`

9.158 Predefined macros

The ARM compiler predefines a number of macros. These macros provide information about toolchain version numbers and compiler options.

The following table 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 9-21 Predefined macros

Name	Value	When defined
`__arm__`	-	Always defined for the ARM compiler, even when you specify the `--thumb` option. See also `__ARMCC_VERSION`.
`__ARMCC_VERSION`	*ver*	Always defined. It is a decimal number, and is guaranteed to increase between releases. The format is *PVVbbbb* where: • *P* is the major version • *VV* is the minor version • *bbbb* is the build number. **Note** Use this macro to distinguish between ARM Compiler 4.1 or later, and other tools that define `__arm__`.
`__APCS_INTERWORK`	-	When you specify the `--apcs /interwork` option or set the target processor architecture to ARMv5T or later.
`__APCS_ROPI`	-	When you specify the `--apcs /ropi` option.
`__APCS_RWPI`	-	When you specify the `--apcs /rwpi` option.
`__APCS_FPIC`	-	When you specify the `--apcs /fpic` option.
`__ARRAY_OPERATORS`	-	In C++ compiler mode, to specify that array new and delete are enabled.
`__BASE_FILE__`	*name*	Always defined. Similar to `__FILE__`, but indicates the primary source file rather than the current one (that is, when the current file is an included file).
`__BIG_ENDIAN`	-	If compiling for a big-endian target.
`_BOOL`	-	In C++ compiler mode, to specify that **`bool`** is a keyword.
`__cplusplus`	-	In C++ compiler mode.
`__CC_ARM`	1	Always set to 1 for the ARM compiler, even when you specify the `--thumb` option.
`__CHAR_UNSIGNED__`	-	In GNU mode. It is defined if and only if **`char`** is an unsigned type.
`__DATE__`	*date*	Always defined.
`__EDG__`	-	Always defined.
`__EDG_IMPLICIT_USING_STD`	-	In C++ mode when you specify the `--using_std` option.

Table 9-21 Predefined macros (continued)

Name	Value	When defined
`__EDG_VERSION__`	-	Always set to an integer value that represents the version number of the *Edison Design Group* (EDG) front-end. For example, version 3.8 is represented as `308`. **Note** The version number of the EDG front-end does not necessarily match the version number of the ARM compiler toolchain.
`__EXCEPTIONS`	1	In C++ mode when you specify the `--exceptions` option.
`__FEATURE_SIGNED_CHAR`	-	When you specify the `--signed_chars` option (used by `CHAR_MIN` and `CHAR_MAX`).
`__FILE__`	*`name`*	Always defined as a string literal.
`__FP_FAST`	-	When you specify the `--fpmode=fast` option.
`__FP_FENV_EXCEPTIONS`	-	When you specify the `--fpmode=ieee_full` or `--fpmode=ieee_fixed` options.
`__FP_FENV_ROUNDING`	-	When you specify the `--fpmode=ieee_full` option.
`__FP_IEEE`	-	When you specify the `--fpmode=ieee_full`, `--fpmode=ieee_fixed`, or `--fpmode=ieee_no_fenv` options.
`__FP_INEXACT_EXCEPTION`	-	When you specify the `--fpmode=ieee_full` option.
`__GNUC__`	*`ver`*	When you specify the `--gnu` option. It is an integer that shows the current major version of the GNU mode being used.
`__GNUC_MINOR__`	*`ver`*	When you specify the `--gnu` option. It is an integer that shows the current minor version of the GNU mode being used.
`__GNUG__`	*`ver`*	In GNU mode when you specify the `--cpp` option. It has the same value as `__GNUC__`.
`__IMPLICIT_INCLUDE`	-	When you specify the `--implicit_include` option.
`__INTMAX_TYPE__`	-	In GNU mode. It defines the correct underlying type for the `intmax_t` **`typedef`**.
`__LINE__`	*`num`*	Always set. It is the source line number of the line of code containing this macro.
`__MODULE__`	*`mod`*	Contains the filename part of the value of `__FILE__`.
`__MULTIFILE`	-	When you explicitly or implicitly use the `--multifile` option.[d]
`__NO_INLINE__`	-	When you specify the `--no_inline` option in GNU mode.
`__OPTIMISE_LEVEL`	*`num`*	Always set to 2 by default, unless you change the optimization level using the `-O`*`num`* option.[d]
`__OPTIMISE_SPACE`	-	When you specify the `-Ospace` option.
`__OPTIMISE_TIME`	-	When you specify the `-Otime` option.
`__OPTIMIZE__`	-	When `-O1`, `-O2`, or `-O3` is specified in GNU mode.

[d] 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 9-21 Predefined macros (continued)

Name	Value	When defined
`__OPTIMIZE_SIZE__`	-	When `-Ospace` is specified in GNU mode.
`__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 `ptrdiff_t` **`typedef`**.
`__RTTI`	-	In C++ mode when RTTI is enabled.
`__sizeof_int`	4	For `sizeof(int)`, but available in preprocessor expressions.
`__sizeof_long`	4	For `sizeof(long)`, but available in preprocessor expressions.
`__sizeof_ptr`	4	For `sizeof(void *)`, but available in preprocessor expressions.
`__SIZE_TYPE__`	-	In GNU mode. It defines the correct underlying type for the `size_t` **`typedef`**.
`__SOFTFP__`	-	If compiling to use the software floating-point calling standard and library. Set when you specify the `--fpu=softvfp` option for ARM or Thumb, or when you specify `--fpu=softvfp+vfpv2` for Thumb.
`__STDC__`	-	In all compiler modes.
`__STDC_VERSION__`	-	Standard version information.
`__STRICT_ANSI__`	-	When you specify the `--strict` option.
`__SUPPORT_SNAN__`	-	Support for signalling NaNs when you specify `--fpmode=ieee_fixed` or `--fpmode=ieee_full`.
`__TARGET_ARCH_ARM`	*`num`*	The number of the ARM base architecture of the target processor irrespective of whether the compiler is compiling for ARM or Thumb. For possible values of `__TARGET_ARCH_ARM` in relation to the ARM architecture versions, see the table below.
`__TARGET_ARCH_THUMB`	*`num`*	The number of the Thumb base architecture of the target processor 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 `__TARGET_ARCH_THUMB` in relation to the ARM architecture versions, see the table below.
`__TARGET_ARCH_`*`XX`*	-	*`XX`* represents the target architecture and its value depends on the target architecture. For example, if you specify the compiler options `--cpu=4T` or `--cpu=ARM7TDMI` then `__TARGET_ARCH_4T` is defined.

Table 9-21 Predefined macros (continued)

Name	Value	When defined
`__TARGET_CPU_`*`XX`*	-	*`XX`* represents the target processor. The value of *`XX`* is derived from the `--cpu` compiler option, or the default if none is specified. For example, if you specify the compiler option `--cpu=ARM7TM` then `__TARGET_CPU_ARM7TM` is defined and no other symbol starting with `__TARGET_CPU_` is defined. If you specify the target architecture, then `__TARGET_CPU_generic` is defined. If the processor name specified with `--cpu` is in lowercase, it is converted to uppercase. For example, `--cpu=Cortex-R4` results in `__TARGET_CPU_CORTEX_R4` being defined (rather than `__TARGET_CPU_Cortex_R4`). If the processor name contains hyphen (-) characters, these are mapped to an underscore (_). For example, `--cpu=ARM1136JF-S` is mapped to `__TARGET_CPU_ARM1136JF_S`.
`__TARGET_FEATURE_DIVIDE`	-	If the target architecture supports the hardware divide instruction (that is, ARMv7-M or ARMv7-R).
`__TARGET_FEATURE_DOUBLEWORD`	-	ARMv5T and above.
`__TARGET_FEATURE_DSPMUL`	-	If the DSP-enhanced multiplier is available, for example ARMv5TE.
`__TARGET_FEATURE_EXTENSION_REGISTER_COUNT`	*`num`*	The number of 64-bit extension registers available in NEON or VFP.
`__TARGET_FEATURE_MULTIPLY`	-	If the target architecture supports the long multiply instructions `MULL` and `MULAL`.
`__TARGET_FEATURE_THUMB`	-	If the target architecture supports Thumb, ARMv4T or later.

Table 9-21 Predefined macros (continued)

Name	Value	When defined
`__TARGET_FPU_`*`xx`*	-	One of the following is set to indicate the FPU usage: • `__TARGET_FPU_NONE` • `__TARGET_FPU_VFP` • `__TARGET_FPU_SOFTVFP` In addition, if compiling with one of the following `--fpu` options, the corresponding target name is set: • `--fpu=softvfp+vfpv2`, `__TARGET_FPU_SOFTVFP_VFPV2` • `--fpu=softvfp+vfpv3`, `__TARGET_FPU_SOFTVFP_VFPV3` • `--fpu=softvfp+vfpv3_fp16`, `__TARGET_FPU_SOFTVFP_VFPV3_FP16` • `--fpu=softvfp+vfpv3_d16`, `__TARGET_FPU_SOFTVFP_VFPV3_D16` • `--fpu=softvfp+vfpv3_d16_fp16`, `__TARGET_FPU_SOFTVFP_VFPV3_D16_FP16` • `--fpu=softvfp+vfpv4`, `__TARGET_FPU_SOFTVFP_VFPV4` • `--fpu=softvfp+vfpv4_d16`, `__TARGET_FPU_SOFTVFP_VFPV4_D16` • `--fpu=softvfp+fpv4-sp`, `__TARGET_FPU_SOFTVFP_FPV4_SP` • `--fpu=vfp`, `__TARGET_FPU_VFPV2` • `--fpu=vfpv2`, `__TARGET_FPU_VFPV2` • `--fpu=vfpv3`, `__TARGET_FPU_VFPV3` • `--fpu=vfpv3_fp16`, `__TARGET_FPU_VFPV3_FP16` • `--fpu=vfpv3_d16`, `__TARGET_FPU_VFPV3_D16` • `--fpu=vfpv3_d16_fp16`, `__TARGET_FPU_VFPV3_D16_FP16` • `--fpu=vfpv4`, `__TARGET_FPU_VFPV4` • `--fpu=vfpv4_d16`, `__TARGET_FPU_VFPV4_D16` • `--fpu=fpv4-sp`, `__TARGET_FPU_FPV4_SP`
`__TARGET_PROFILE_R`		When you specify the `--cpu=7-R` option.
`__TARGET_PROFILE_M`		When you specify any of the following options: • `--cpu=6-M` • `--cpu=6S-M` • `--cpu=7-M`
`__thumb__`	-	When the compiler is in Thumb state. That is, you have either specified the `--thumb` option on the command-line or `#pragma thumb` in your source code. **Note** • The compiler might generate some ARM code even if it is compiling for Thumb. • `__thumb` and `__thumb__` become defined or undefined when using `#pragma thumb` or `#pragma arm`, but do not change in cases where Thumb functions are generated as ARM code for other reasons (for example, a function specified as `__irq`).
`__TIME__`	*`time`*	Always defined.

Table 9-21 Predefined macros (continued)

Name	Value	When defined
`__UINTMAX_TYPE__`	-	In GNU mode. It defines the correct underlying type for the `uintmax_t` **`typedef`**.
`__USER_LABEL_PREFIX__`		In GNU mode. It defines an empty string.
`__VERSION__`	*`ver`*	When you specify the `--gnu` option. It is a string that shows the current version of the GNU mode being used.
`_WCHAR_T`	-	In C++ mode, to specify that **`wchar_t`** is a keyword.
`__WCHAR_TYPE__`	-	In GNU mode. It defines the correct underlying type for the `wchar_t` **`typedef`**.
`__WCHAR_UNSIGNED__`	-	In GNU mode when you specify the `--cpp` option. It is defined if and only if **`wchar_t`** is an unsigned type.
`__WINT_TYPE__`	-	In GNU mode. It defines the correct underlying type for the `wint_t` **`typedef`**.

The following table shows the possible values for `__TARGET_ARCH_THUMB`, and how these values relate to versions of the ARM architecture.

Table 9-22 Thumb architecture versions in relation to ARM architecture versions

ARM architecture	`__TARGET_ARCH_ARM`	`__TARGET_ARCH_THUMB`
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-R	`7`	`4`
v7-M, v7E-M	`0`	`4`

Related references

9.159 Built-in function name variables on page 9-703.

9.159 Built-in function name variables

The following table lists built-in variables that the compiler supports for C and C++.

Table 9-23 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.

Related references

9.158 Predefined macros on page 9-697.

Chapter 10
C and C++ Implementation Details

Describes the language implementation details for the compiler. Some language implementation details are common to both C and C++, while others are specific to C++.

It contains the following sections:

- *10.1 Character sets and identifiers in ARM C and C++* on page 10-705.
- *10.2 Basic data types in ARM C and C++* on page 10-707.
- *10.3 Operations on basic data types ARM C and C++* on page 10-709.
- *10.4 Structures, unions, enumerations, and bitfields in ARM C and C++* on page 10-710.
- *10.5 Using the ::operator new function in ARM C++* on page 10-715.
- *10.6 Tentative arrays in ARM C++* on page 10-716.
- *10.7 Old-style C parameters in ARM C++ functions* on page 10-717.
- *10.8 Anachronisms in ARM C++* on page 10-718.
- *10.9 Template instantiation in ARM C++* on page 10-719.
- *10.10 Namespaces in ARM C++* on page 10-720.
- *10.11 C++ exception handling in ARM C++* on page 10-722.
- *10.12 Extern inline functions in ARM C++* on page 10-723.
- *10.13 C++11 supported features* on page 10-724.

10.1 Character sets and identifiers in ARM C and C++

Describes the character set and identifier implementation details in ARM C and C++.

The following point applies to the 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.

The following points apply to the character sets expected by the compiler:

- 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 change the locale using the `--locale` command-line option if the source file contains non-ASCII characters. If you do not specify `--locale`, the system locale is used.
- The compiler supports multibyte character sets, such as Unicode. You can control this support using the `--[no_]multibyte_chars` options.
- If the source file encoding is UTF-8 or UTF-16, and the file starts with a byte order mark then the compiler ignores the `--[no_]multibyte_chars` and `--locale` options and interprets the file as UTF-8 or UTF-16.
- 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 `NUL` (`\0`) character.
- All integer character constants that contain a single character, or character escape sequence, are represented in both the source and execution character sets.The following table lists the supported character escape codes.

Table 10-1 Character escape codes

Escape sequence	Char value	Description
`\a`	`7`	Attention (bell)
`\b`	`8`	Backspace
`\t`	`9`	Horizontal tab
`\n`	`10`	New line (line feed)
`\v`	`11`	Vertical tab
`\f`	`12`	Form feed

Table 10-1 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 makes the **`char`** signed
  — the `--unsigned_chars` option makes 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.

- Converting multibyte characters into the corresponding wide characters for a wide character constant does not use a locale. This is not relevant to the generic implementation.

10.2 Basic data types in ARM C and C++

Describes the basic data types implemented in ARM C and C++:

Size and alignment of basic data types

The following table gives the size and natural alignment of the basic data types.

Table 10-2 Size and alignment of data types

Type	Size in bits	Natural alignment in bytes	Range of values
`char`	8	1 (byte-aligned)	0 to 255 (unsigned) by default. –128 to 127 (signed) when compiled with `--signed_chars`.
`signed char`	8	1 (byte-aligned)	–128 to 127
`unsigned char`	8	1 (byte-aligned)	0 to 255
`(signed) short`	16	2 (halfword-aligned)	–32,768 to 32,767
`unsigned short`	16	2 (halfword-aligned)	0 to 65,535
`(signed) int`	32	4 (word-aligned)	–2,147,483,648 to 2,147,483,647
`unsigned int`	32	4 (word-aligned)	0 to 4,294,967,295
`(signed) long`	32	4 (word-aligned)	–2,147,483,648 to 2,147,483,647
`unsigned long`	32	4 (word-aligned)	0 to 4,294,967,295
`(signed) long long`	64	8 (doubleword-aligned)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long long`	64	8 (doubleword-aligned)	0 to 18,446,744,073,709,551,615
`float`	32	4 (word-aligned)	1.175494351e-38 to 3.40282347e+38 (normalized values)
`double`	64	8 (doubleword-aligned)	2.22507385850720138e-308 to 1.79769313486231571e +308 (normalized values)
`long double`	64	8 (doubleword-aligned)	2.22507385850720138e-308 to 1.79769313486231571e +308 (normalized values)
`wchar_t`	16 32	2 (halfword-aligned) 4 (word-aligned)	0 to 65,535 by default. 0 to 4,294,967,295 when compiled with `--wchar32`.
All pointers	32	4 (word-aligned)	Not applicable.
`bool` (C++ only)	8	1 (byte-aligned)	false or true
`_Bool` (C only[e])	8	1 (byte-aligned)	false or true

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.

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.

[e] `stdbool.h` lets you define the `bool` macro in C.

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.

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`.

10.3 Operations on basic data types ARM C and C++

Describes the basic data type arithmetic operation implementation details in ARM C and C++.

The ARM compiler performs the usual arithmetic conversions set out in relevant sections of the ISO C99 and ISO C++ standards. The following information describes additional points that relate to arithmetic operations.

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 required 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.

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.

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`.

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:

  ```
  ((int)a - (int)b) / (int)sizeof(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 **`struct`**s, both pointers must point to elements of the same array.

10.4 Structures, unions, enumerations, and bitfields in ARM C and C++

Describes the implementation of the structured data types `union`, `enum`, and `struct`. It also describes structure padding and bitfield implementation.

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.

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 was 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.

In strict C, enumerator values must be representable as **`int`**s. That is, they must be in the range -2147483648 to +2147483647, inclusive. 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 ...
```

Structures

The following points apply to:

- all C structures
- all C++ structures and classes not using virtual functions or base classes.

Structures can contain padding to ensure that fields are correctly aligned and that the structure itself is correctly aligned. The following diagram 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.

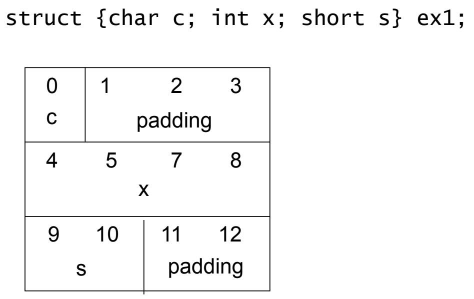

Figure 10-1 Conventional nonpacked structure example

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.

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.

Bitfields

In nonpacked structures, 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. The following figure shows the bitfield allocation for the following example structure:

```
struct X
{
    int x:10;
    char y:8;
};
```

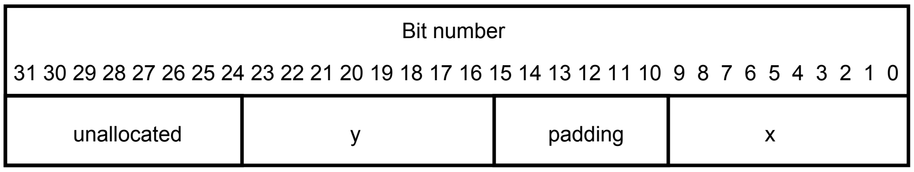

Figure 10-2 Bitfield allocation 1

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, as follows:

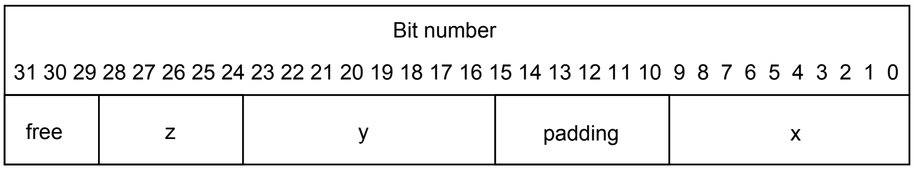

Figure 10-3 Bitfield allocation 2

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.

Note

As an optimization, the compiler might overwrite padding bits in a container with unspecified values when a bitfield is written. This does not affect normal usage of bitfields.

Packing and alignment of bitfields

Using `__attribute__((aligned(n)))` makes the bitfield member n-byte aligned, not just its container, but the bitfield is aligned to the packed alignment at most. It is ignored on bitfields in `__packed` and `__attribute__((packed))` structs.

The alignment of a bitfield member's container is the same as the alignment of that bitfield member. The size of a bitfield container is the least multiple of the alignment that fully covers the bitfield, but no larger than the size of the container-type. The following code examples show this:

```
#pragma pack(2)
/* The container of b must start at a 2 byte alignment boundary, and must
* have a size no larger than the container type, in this case the size of
* a short. The container of b cannot start at offset 0 (overlapping with a)
* as the bitfield b would then start at offset 2, and would not fully lie
* within the container. Therefore the container for b must start at offset
* 2.
*
* Data layout:      0x11 0x00 0x22 0x22
* Container layout:| a  |    |    b    |
*/
struct {
  char a;
  short b : 16;
} var1 = { 0x11, 0x2222 };


/* The container of b can be up to 4 bytes. Its size must be either 2
* or 4 bytes, as these are the multiples of the alignment that are no
* larger than the size of the container type. With a 4 byte container
* starting at 0, the bitfield b can start at offset 1 and fully lie
* within the container.
*
* Data layout:      0x11 0x22 0x22 0x00
* Container layout:| a  |
*                  |        b         |
*/
struct {
  char a;
  int b : 16;
} var2 = { 0x11, 0x2222 };
```

Packed bitfield containers, including all bitfield containers in packed structures, have an alignment of 1. Therefore the maximum bit padding inserted to align a packed bitfield container is 7 bits.

For an unpacked bitfield container, the maximum bit padding is `8*sizeof(container-type)-1` bits.

Tail-padding is always inserted into the structure as necessary to ensure arrays of the structure have their elements correctly aligned.

A packed bitfield container is only large enough (in bytes) to hold the bitfield that declared it. Non-packed bitfield containers are the size of their type.

The following examples illustrate these interactions.

```
struct A {          int z:17; }; // sizeof(A) = 4, alignment = 4
struct A { __packed int z:17; }; // sizeof(A) = 3, alignment = 1
__packed struct A { int z:17; }; // sizeof(A) = 3, alignment = 1
```

```
struct A { char y:1;          int z:31; }; // sizeof(A) = 4, alignment = 4
struct A { char y:1; __packed int z:31; }; // sizeof(A) = 4, alignment = 1
__packed struct A { char y:1; int z:31; }; // sizeof(A) = 4, alignment = 1
```

```
struct A { char y:1;          int z:32; }; // sizeof(A) = 8, alignment = 4
struct A { char y:1; __packed int z:32; }; // sizeof(A) = 5, alignment = 1
__packed struct A { char y:1; int z:32; }; // sizeof(A) = 5, alignment = 1
```

```
struct A { int x; char y:1;          int z:31; };  // sizeof(A) = 8, alignment = 4
struct A { int x; char y:1; __packed int z:31; };  // sizeof(A) = 8, alignment = 4
__packed struct A { int x; char y:1; int z:31; };  // sizeof(A) = 8, alignment = 1
```

```
struct A { int x; char y:1;          int z:32; };  // sizeof(A) = 12, alignment = 4 [1]
struct A { int x; char y:1; __packed int z:32; };  // sizeof(A) = 12, alignment = 4 [2]
__packed struct A { int x; char y:1; int z:32; };  // sizeof(A) = 9, alignment = 1
```

Note that [1] and [2] are not identical; the location of z within the structure and the tail-padding differ.

```
struct example1
{
int a : 8;  /* 4-byte container at offset 0 */
__packed int b : 8;  /* 1-byte container at offset 1 */
__packed int c : 24; /* 3-byte container at offset 2 */
}; /* Total size 8 (3 bytes tail padding) */;
```

```
struct example2
{
__packed int a : 8; /* 1-byte container at offset 0 */
__packed int b : 8; /* 1-byte container at offset 1 */
int c : 8; /* 4-byte container at offset 0 */
}; /* Total size 4 (No tail padding) */
```

```
struct example3
{
int a : 8;  /* 4-byte container at offset 0 */
__packed int b : 32; /* 4-byte container at offset 1 */
__packed int c : 32; /* 4-byte container at offset 5 */
int d : 16; /* 4-byte container at offset 8 */
int e : 16; /* 4-byte container at offset 12 */
int f : 16; /* In previous container */
}; /* Total size 16 (No tail padding) */
```

Related references

9.12 __packed on page 9-527.

9.58 __attribute__((packed)) type attribute on page 9-577.

9.66 __attribute__((packed)) variable attribute on page 9-585.

9.95 #pragma pack(n) on page 9-615.

10.5 Using the ::operator new function in ARM C++

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`.

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.

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.

10.6 Tentative arrays in ARM C++

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.

10.7 Old-style C parameters in ARM 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.

10.8 Anachronisms in ARM C++

You can enable support for anachronisms using the `--anachronisms` option.

The following anachronisms are accepted:

- **`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.
- You can overload both prefix and postfix operations with a single `operator++()` and `operator--()` function.
- 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.

Related references

7.5 --anachronisms, --no_anachronisms on page 7-272.

10.9 Template instantiation in ARM C++

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.

————————————

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.

10.10 Namespaces in ARM C++

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 required 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.

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.

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.

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);
```

10.11 C++ exception handling in ARM C++

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.

Note

The Rogue Wave Standard C++ Library is provided with C++ exceptions enabled.

You can exercise limited control over exception table generation.

Function unwinding at runtime

By default, functions compiled with `--exceptions` can be unwound at runtime. *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`. 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.
- The exception table representation that describes the function is very compact. This assists smart linkers with table optimization.
- Function inlining is restricted, because the caller and callee must interact correctly.

Therefore, `#pragma no_exceptions_unwind` lets you 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.

Related references

7.59 --exceptions_unwind, --no_exceptions_unwind on page 7-333.

7.58 --exceptions, --no_exceptions on page 7-332.

9.83 #pragma exceptions_unwind, #pragma no_exceptions_unwind on page 9-603.

7.59 --exceptions_unwind, --no_exceptions_unwind on page 7-333.

10.12 Extern inline functions in ARM C++

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.

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 that is used by the compiler decides that it is better not to inline the function. `-Ospace` and `-Otime` influence the heuristic. 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;
}
```

Note

__forceinline does not guarantee the function is inlined, since the decision ultimately lies with the compiler. Using __forceinline provides a hint, telling the compiler to inline the function if possible.

10.13 C++11 supported features

ARM Compiler supports a large subset of the language features of C++11.

Fully supported C++11 features

ARM Compiler fully supports the following language features as defined by the C++11 language standard:

- `auto` can be used as a type specifier in declaration of a variable or reference.
- `constexpr`.
- Trailing return types are allowed in top-level function declarators.
- Variadic templates.
- Alias and alias template declarations such as `using X = int`.
- Support for double right angle bracket token `>>` interpreted as template argument list termination.
- `static_assert`.
- Scoped enumerations with `enum` classes.
- Unrestricted unions.
- Extended `friend` class syntax extensions.
- `noexcept` operator and specifier.
- Non-static data member initializers.
- Local and unnamed types can be used for template type arguments.
- Use of `extern` keyword to suppress an explicit template instantiation.
- Keyword `final` on class types and virtual functions.
- Keyword `override` can be used on virtual functions.
- Generation of move constructor and move assignment operator special member functions.
- Functions can be deleted with `=delete`.
- Raw and UTF-8 string literals.
- Support for `char16_t` and `char32_t` character types and `u` and `U` string literals.
- C99 language features accepted by the C++11 standard.
- Type conversion functions can be marked `explicit`.
- Inline namespaces.
- Support for expressions in template deduction contexts.

Partially supported C++11 features with restrictions

ARM Compiler partially supports the following language features. You can use the feature, but restrictions might apply.

- `nullptr`.

  ARM Compiler supports the keyword `nullptr`. However, the standard library header file does not contain a definition of `std::nullptr_t`. It can be defined manually using:

  ```
  namespace std
  {
      typedef decltype(nullptr) nullptr_t;
  }
  ```
- Rvalue references.

  ARM Compiler supports rvalue references and reference qualified member functions. However, the standard library provided with ARM Compiler does not come with an implementation of `std::move` or `std::forward`.

  Both `std::move` and `std::forward` are a `static_cast` with the target type deduced via template argument deduction. An example implementation is as follows:

  ```
  namespace std
  {
      template< class T > struct remove_reference      {typedef T type;};
      template< class T > struct remove_reference<T&>  {typedef T type;};
      template< class T > struct remove_reference<T&&> {typedef T type;};

      template<class T>
      typename remove_reference<T>::type&&
  ```

```
    move(T&& a) noexcept
    {
        typedef typename remove_reference<T>::type&& RvalRef;
        return static_cast<RvalRef>(a);
    }
    template<class T>
    T&&
    forward(typename remove_reference<T>::type& a) noexcept
    {
        return static_cast<T&&>(a);
    }
}
```

ARM Compiler does not implement the C++11 value categories such as prvalue, xvalue and glvalue as described in *A Taxonomy of Expression Value Categories*. Instead it implements the draft C++0x definitions of `lvalue` and `rvalue`. In rare cases this may result in some differences in behavior from the C++11 standard when returning `rvalue` references from functions.

- Initializer lists and uniform initialization.

  ARM Compiler supports initializer lists and uniform initialization, but the standard library does not provide an implementation of `std::initializer_list`. With a user-supplied implementation of `std::initializer_list` initializer lists and uniform initialization can be used.
- Lambda functions

  ARM Compiler supports lambda functions. The standard library provided with the ARM Compiler does not provide an implementation of `std::function`. This means that lambda functions can only be used when type deduction is used.

  Within a function `auto` can be used to store the generated lambda function, a lambda can also be passed as a parameter to a function template, for example:

```
#include <iostream>
template<typename T> void call_lambda(T lambda)
{
    lambda();
}
void function()
{
    auto lambda = [] () { std::cout << “Hello World”; };
    call_lambda(lambda);
}
```

- Range-based `for` loops.

  ARM Compiler supports range-based `for` loops. However an implementation of `std::initializer_list` is required for range-based `for` loops with braced initializers, for example:

```
for(auto x : {1,2,3}) { std::cout << x << std::endl; }
```

- Decltype
  The `decltype` operator is supported, but does not include the C++11 extensions N3049 and N3276. This means that `decltype` cannot be used in all places allowed by the standard. In summary the following uses of `decltype` are not supported:
  — As a name qualifier, for example `decltype(x)::count`.
  — In destructor calls, for example `p->~decltype(x)();`.
  — As a base specifier, for example `class X : decltype(Y) {};`.
  — `decltype` construct cannot be a call to a function that has an incomplete type.
- C++11 attribute syntax

  ARM Compiler supports the `[[noreturn]]` attribute.

  ARM Compiler ignores the `[[carries_dependency]]` attribute.
- Delegating constructors

  Delegating constructors are supported by the compiler. However when exceptions are enabled you must link against up-to-date C++ runtime libraries.
- Support of `=default` for special member functions

Special member functions can be explicitly given default implementation with `=default`. ARM Compiler does not support this for the move constructor or move assignment operator special member functions. All other special member functions are supported. For example:

```
struct X
{
    // The constructor, destructor, copy constructor
    // and copy assignment operator are supported
    X() = default;
    ~X() = default;
    X(const X&) = default;
    X& operator=(const X&) = default;

    // The move constructor and move assignment operator are not supported
    X(const X&&) = default;
    X& operator=(const X&&) = default;
};
```

Unsupported C++11 features

The following language features are not supported in any way by ARM Compiler:

- C++11 memory model guarantees for `std::atomic`.
- The `alignof` operator and `alignas` specifier.
- Inheriting constructors.
- Thread local storage keyword `thread_local`.
- User-defined literals.
- Smart pointers.

Note

The C++ libraries provided with ARM Compiler 5 do not support C++11.

Chapter 11
What is Semihosting?

Describes the semihosting mechanism.

It contains the following sections:

- *11.1 What is semihosting?* on page 11-729.
- *11.2 The semihosting interface* on page 11-730.
- *11.3 Can I change the semihosting operation numbers?* on page 11-731.
- *11.4 Debug agent interaction SVCs* on page 11-732.
- *11.5 angel_SWIreason_EnterSVC (0x17)* on page 11-733.
- *11.6 angel_SWIreason_ReportException (0x18)* on page 11-734.
- *11.7 SYS_CLOSE (0x02)* on page 11-736.
- *11.8 SYS_CLOCK (0x10)* on page 11-737.
- *11.9 SYS_ELAPSED (0x30)* on page 11-738.
- *11.10 SYS_ERRNO (0x13)* on page 11-739.
- *11.11 SYS_FLEN (0x0C)* on page 11-740.
- *11.12 SYS_GET_CMDLINE (0x15)* on page 11-741.
- *11.13 SYS_HEAPINFO (0x16)* on page 11-742.
- *11.14 SYS_ISERROR (0x08)* on page 11-743.
- *11.15 SYS_ISTTY (0x09)* on page 11-744.
- *11.16 SYS_OPEN (0x01)* on page 11-745.
- *11.17 SYS_READ (0x06)* on page 11-746.
- *11.18 SYS_READC (0x07)* on page 11-747.
- *11.19 SYS_REMOVE (0x0E)* on page 11-748.
- *11.20 SYS_RENAME (0x0F)* on page 11-749.
- *11.21 SYS_SEEK (0x0A)* on page 11-750.
- *11.22 SYS_SYSTEM (0x12)* on page 11-751.
- *11.23 SYS_TICKFREQ (0x31)* on page 11-752.

- *11.24 SYS_TIME (0x11)* on page 11-753.
- *11.25 SYS_TMPNAM (0x0D)* on page 11-754.
- *11.26 SYS_WRITE (0x05)* on page 11-755.
- *11.27 SYS_WRITEC (0x03)* on page 11-756.
- *11.28 SYS_WRITE0 (0x04)* on page 11-757.

11.1 What is semihosting?

Semihosting is a mechanism that enables code running on an ARM target to communicate and use the Input/Output facilities on a host computer that is running a debugger.

Examples of these facilities include keyboard input, screen output, and disk I/O. For example, you can use this mechanism to enable functions in the C library, such as `printf()` and `scanf()`, to use the screen and keyboard of the host instead of having a screen and keyboard on the target system.

This is useful because development hardware often does not have all the input and output facilities of the final system. Semihosting enables the host computer to provide these facilities.

Semihosting is implemented by a set of defined software instructions, for example SVCs, that generate exceptions from program control. The application invokes the appropriate semihosting call and the debug agent then handles the exception. The debug agent provides the required communication with the host.

The semihosting interface is common across all debug agents provided by ARM. Semihosted operations work when you are debugging applications on your development platform, as shown in the following figure:

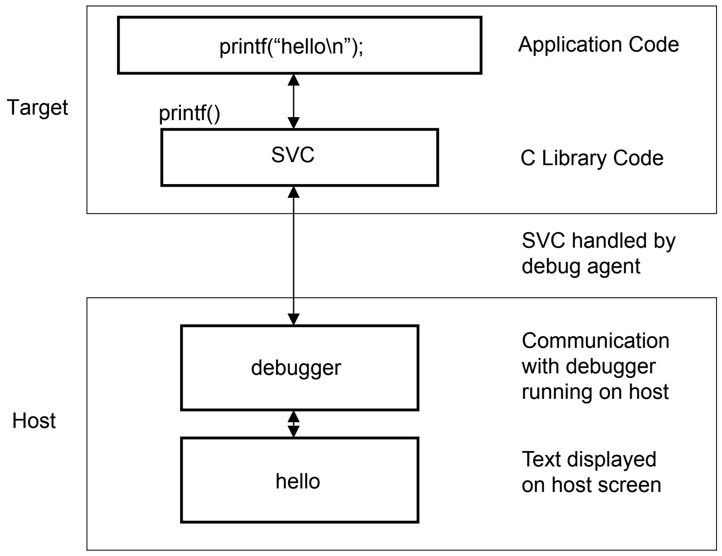
printf("hello\n");
Application Code
Target
printf()
SVC
C Library Code
SVC handled by debug agent
debugger
Communication with debugger running on host
Host
hello
Text displayed on host screen

Figure 11-1 Semihosting overview

In many cases, semihosting is invoked by code within library functions. The application can also invoke the semihosting operation directly.

Note

ARM processors use the `SVC` instructions, formerly known as `SWI` instructions, to make semihosting calls. However, if you are compiling for an ARMv6-M or ARMv7-M, for example a Cortex-M1 or Cortex-M3 processor, semihosting is implemented using the `BKPT` instruction.

Related concepts

11.2 The semihosting interface on page 11-730.

11.3 Can I change the semihosting operation numbers? on page 11-731.

11.4 Debug agent interaction SVCs on page 11-732.

Related information

The ARM C and C++ libraries.

11.2 The semihosting interface

The ARM and Thumb `SVC` instructions contain a field that encodes the SVC number used by the application code.

Note

If you are compiling for the ARMv6-M or ARMv7-M, the Thumb `BKPT` instruction is used instead of the Thumb `SVC` instruction. Both `BKPT` and `SVC` take an 8-bit immediate value. In all other respects, semihosting is the same for all supported ARM processors.

The system SVC handler can decode the SVC number. Semihosting operations are requested using a single SVC number, leaving the other numbers available for use by the application or operating system. The SVC number used for semihosting depends on the target architecture or processor:

`SVC 0x123456`
: In ARM state for all architectures.

`SVC 0xAB`
: In ARM state and Thumb state, excluding ARMv6-M and ARMv7-M. This behavior is not guaranteed on all debug targets from ARM or from third parties.

`BKPT 0xAB`
: For ARMv6-M and ARMv7-M, Thumb state only.

The SVC number indicates to the debug agent that the `SVC` instruction is a semihosting request. To distinguish between operations, the operation type is passed in `R0`. All other parameters are passed in a block that is pointed to by `R1`.

The result is returned in `R0`, either as an explicit return value or as a pointer to a data block. Even if no result is returned, assume that `R0` is corrupted.

The available semihosting operation numbers passed in `R0` are allocated as follows:

`0x00-0x31`
: Used by ARM.

`0x32-0xFF`
: Reserved for future use by ARM.

`0x100-0x1FF`
: Reserved for user applications. These are not used by ARM.

  If you are writing your own SVC operations, however, you are advised to use a different SVC number rather than using the semihosted SVC number and these operation type numbers.

`0x200-0xFFFFFFFF`
: Undefined and currently unused. It is recommended that you do not use these.

In the following sections, the number in parentheses after the operation name is the value placed into `R0`, for example `SYS_OPEN` (`0x01`).

If you are calling SVCs from assembly language code ARM recommends that you define the semihosting operation names, to their respective operation numbers, with the `EQU` directive. For example:

```
SYS_OPEN    EQU 0x01
SYS_CLOSE   EQU 0x02
```

Related concepts

11.1 What is semihosting? on page 11-729.

11.3 Can I change the semihosting operation numbers? on page 11-731.

11.4 Debug agent interaction SVCs on page 11-732.

11.3 Can I change the semihosting operation numbers?

ARM strongly recommends that you do not change the semihosting operation numbers.

However, if you have to do this, you must:

- change all the code in your system, including library code, to use the new number
- reconfigure your debugger to use the new number.

Related concepts

11.1 What is semihosting? on page 11-729.

11.2 The semihosting interface on page 11-730.

11.4 Debug agent interaction SVCs

In addition to the C library semihosted functions, some other SVCs support interaction with the debug agent.

These are:

- `angel_SWIreason_EnterSVC` (0x17)
- `angel_SWIreason_ReportException` (0x18).

Related references

11.5 angel_SWIreason_EnterSVC (0x17) on page 11-733.

11.6 angel_SWIreason_ReportException (0x18) on page 11-734.

11.5 angel_SWIreason_EnterSVC (0x17)

Sets the processor to Supervisor mode and disables all interrupts by setting both interrupt mask bits in the new `CPSR`.

With a debug hardware unit, such as ARM RVI™ debug unit or ARM DSTREAM™ debug and trace unit:

- the User stack pointer, `R13_USR`, is copied to the Supervisor mode stack pointer, `R13_SVC`
- the I and F bits in the current `CPSR` are set, which disables normal and fast interrupts.

Entry

Register `R1` is not used. The `CPSR` can specify User or Supervisor mode.

Return

On exit, `R0` contains the address of a function to be called to return to User mode. The function has the following prototype:

```
void ReturnToUSR(void)
```

If `EnterSVC` is called in User mode, this routine returns the caller to User mode and restores the interrupt flags. Otherwise, the action of this routine is undefined.

If entered in User mode, the Supervisor mode stack is lost as a result of copying the user stack pointer. The return to User routine restores `R13_SVC` to the Supervisor mode stack value, but this stack must not be used by applications.

After executing the SVC, the current link register is `R14_SVC`, not `R14_USR`. If the value of `R14_USR` is required after the call, it must be pushed onto the stack before the call and popped afterwards, as for a `BL` function call.

11.6 angel_SWIreason_ReportException (0x18)

This SVC can be called by an application to report an exception to the debugger directly. The most common use is to report that execution has completed, using `ADP_Stopped_ApplicationExit`.

Entry

On entry `R1` is set to one of the values listed in the following tables. These values are defined in `angel_reasons.h`.

The hardware exceptions are generated if the debugger variable `vector_catch` is set to catch that exception type, and the debug agent is capable of reporting that exception type. The following table shows the hardware vector reason codes:

Table 11-1 Hardware vector reason codes

Name	Hexadecimal value
`ADP_Stopped_BranchThroughZero`	`0x20000`
`ADP_Stopped_UndefinedInstr`	`0x20001`
`ADP_Stopped_SoftwareInterrupt`	`0x20002`
`ADP_Stopped_PrefetchAbort`	`0x20003`
`ADP_Stopped_DataAbort`	`0x20004`
`ADP_Stopped_AddressException`	`0x20005`
`ADP_Stopped_IRQ`	`0x20006`
`ADP_Stopped_FIQ`	`0x20007`

Exception handlers can use these SVCs at the end of handler chains as the default action, to indicate that the exception has not been handled. The following table shows the software reason codes:

Table 11-2 Software reason codes

Name	Hexadecimal value
`ADP_Stopped_BreakPoint`	`0x20020`
`ADP_Stopped_WatchPoint`	`0x20021`
`ADP_Stopped_StepComplete`	`0x20022`
`ADP_Stopped_RunTimeErrorUnknown`	`*0x20023`
`ADP_Stopped_InternalError`	`*0x20024`
`ADP_Stopped_UserInterruption`	`0x20025`
`ADP_Stopped_ApplicationExit`	`0x20026`
`ADP_Stopped_StackOverflow`	`*0x20027`
`ADP_Stopped_DivisionByZero`	`*0x20028`
`ADP_Stopped_OSSpecific`	`*0x20029`

In this table, a * next to a value indicates that the value is not supported by the ARM debugger. The debugger reports an `Unhandled ADP_Stopped exception` for these values.

Return

No return is expected from these calls. However, it is possible for the debugger to request that the application continue by performing an `RDI_Execute` request or equivalent. In this case, execution continues with the registers as they were on entry to the SVC, or as subsequently modified by the debugger.

11.7 SYS_CLOSE (0x02)

Closes a file on the host system. The handle must reference a file that was opened with `SYS_OPEN`.

Entry

On entry, `R1` contains a pointer to a one-word argument block:

word 1
contains a handle for an open file.

Return

On exit, `R0` contains:

- 0 if the call is successful
- –1 if the call is not successful.

Related references

11.16 SYS_OPEN (0x01) on page 11-745.

11.8 SYS_CLOCK (0x10)

Returns the number of centiseconds since the execution started.

Values returned by this SVC can be of limited use for some benchmarking purposes because of communication overhead or other agent-specific factors. For example, with a debug hardware unit such as RVI or DSTREAM, the request is passed back to the host for execution. This can lead to unpredictable delays in transmission and process scheduling.

Use this function to calculate time intervals, by calculating differences between intervals with and without the code sequence to be timed.

Entry

Register `R1` must contain zero. There are no other parameters.

Return

On exit, `R0` contains:

- the number of centiseconds since some arbitrary start point, if the call is successful
- –1 if the call is not successful, for example, because of a communications error.

Related references

11.9 SYS_ELAPSED (0x30) on page 11-738.

11.23 SYS_TICKFREQ (0x31) on page 11-752.

11.9 SYS_ELAPSED (0x30)

Returns the number of elapsed target ticks since execution started.

Use `SYS_TICKFREQ` to determine the tick frequency.

Entry

On entry, `R1` points to a two-word data block to be used for returning the number of elapsed ticks:

word 1
the least significant word and is at the low address

word 2
the most significant word and is at the high address.

Return

On exit:

- On success, `R1` points to a doubleword that contains the number of elapsed ticks. On failure, `R1` contains -1.
- On success, `R0` contains 0. On failure, `R0` contains -1.

Note

Some debuggers might not support this SVC when connected though RVI or DSTREAM, and they always return –1 in `R0`.

11.10 SYS_ERRNO (0x13)

Returns the value of the C library `errno` variable associated with the host implementation of the semihosting SVCs.

The `errno` variable can be set by a number of C library semihosted functions, including:

- `SYS_REMOVE`
- `SYS_OPEN`
- `SYS_CLOSE`
- `SYS_READ`
- `SYS_WRITE`
- `SYS_SEEK`.

Whether `errno` is set or not, and to what value, is entirely host-specific, except where the ISO C standard defines the behavior.

Entry

There are no parameters. Register `R1` must be zero.

Return

On exit, `R0` contains the value of the C library `errno` variable.

Related references

11.7 SYS_CLOSE (0x02) on page 11-736.
11.16 SYS_OPEN (0x01) on page 11-745.
11.17 SYS_READ (0x06) on page 11-746.
11.19 SYS_REMOVE (0x0E) on page 11-748.
11.21 SYS_SEEK (0x0A) on page 11-750.
11.26 SYS_WRITE (0x05) on page 11-755.

11.11 SYS_FLEN (0x0C)

Returns the length of a specified file.

Entry

On entry, `R1` contains a pointer to a one-word argument block:

word 1
A handle for a previously opened, seekable file object.

Return

On exit, `R0` contains:

- the current length of the file object, if the call is successful
- –1 if an error occurs.

11.12 SYS_GET_CMDLINE (0x15)

Returns the command line used for the call to the executable, that is, `argc` and `argv`.

Entry

On entry, `R1` points to a two-word data block to be used for returning the command string and its length:

word 1
: a pointer to a buffer of at least the size specified in word two

word 2
: the length of the buffer in bytes.

Return

On exit:

- Register `R1` points to a two-word data block:The debug agent might impose limits on the maximum length of the string that can be transferred. However, the agent must be able to transfer a command line of at least 80 bytes.

  word 1
  : a pointer to null-terminated string of the command line

  word 2
  : the length of the string.
- Register `R0` contains an error code:
  - 0 if the call is successful
  - –1 if the call is not successful, for example, because of a communications error.

11.13 SYS_HEAPINFO (0x16)

Returns the system stack and heap parameters.

The values returned are typically those used by the C library during initialization. For a debug hardware unit, such as RVI or DSTREAM, the values returned are the image location and the top of memory.

The C library can override these values.

The host debugger determines the actual values to return by using the `top_of_memory` debugger variable.

Entry

On entry, `R1` contains the address of a pointer to a four-word data block. The contents of the data block are filled by the function. The following example shows the structure of the data block and return values.

```
struct block {
    int heap_base;
    int heap_limit;
    int stack_base;
    int stack_limit;
};
struct block *mem_block, info;
mem_block = &info;
AngelSWI(SYS_HEAPINFO, (unsigned) &mem_block);
```

Note

If word one of the data block has the value zero, the C library replaces the zero with `Image$$ZI$$Limit`. This value corresponds to the top of the data region in the memory map.

Return

On exit, `R1` contains the address of the pointer to the structure.

If one of the values in the structure is 0, the system was unable to calculate the real value.

11.14 SYS_ISERROR (0x08)

Determines whether the return code from another semihosting call is an error status or not.

This call is passed a parameter block containing the error code to examine.

Entry

On entry, `R1` contains a pointer to a one-word data block:

word 1
The required status word to check.

Return

On exit, `R0` contains:

- 0 if the status word is not an error indication
- a nonzero value if the status word is an error indication.

11.15 SYS_ISTTY (0x09)

Checks whether a file is connected to an interactive device.

Entry

On entry, `R1` contains a pointer to a one-word argument block:

word 1
A handle for a previously opened file object.

Return

On exit, `R0` contains:

- 1 if the handle identifies an interactive device
- 0 if the handle identifies a file
- a value other than 1 or 0 if an error occurs.

11.16 SYS_OPEN (0x01)

Opens a file on the host system.

The file path is specified either as relative to the current directory of the host process, or absolute, using the path conventions of the host operating system.

ARM targets interpret the special path name `:tt` as meaning the console input stream, for an open-read or the console output stream, for an open-write. Opening these streams is performed as part of the standard startup code for those applications that reference the C `stdio` streams.

Entry

On entry, `R1` contains a pointer to a three-word argument block:

word 1
A pointer to a null-terminated string containing a file or device name.

word 2
An integer that specifies the file opening mode. The following table gives the valid values for the integer, and their corresponding ISO C `fopen()` mode.

word 3
An integer that gives the length of the string pointed to by word 1.

The length does not include the terminating null character that must be present.

Table 11-3 Value of mode

mode	0	1	2	3	4	5	6	7	8	9	10	11
ISO C fopen mode[f]	r	rb	r+	r+b	w	wb	w+	w+b	a	ab	a+	a+b

Return

On exit, `R0` contains:
- a nonzero handle if the call is successful
- –1 if the call is not successful.

f The non-ANSI option t is not supported.

11.17 SYS_READ (0x06)

Reads the contents of a file into a buffer.

The file position is specified either:

- explicitly by a `SYS_SEEK`
- implicitly one byte beyond the previous `SYS_READ` or `SYS_WRITE` request.

The file position is at the start of the file when it is opened, and is lost when the file is closed. Perform the file operation as a single action whenever possible. For example, do not split a read of 16KB into four 4KB chunks unless there is no alternative.

Entry

On entry, `R1` contains a pointer to a four-word data block:

word 1
contains a handle for a file previously opened with `SYS_OPEN`

word 2
points to a buffer

word 3
contains the number of bytes to read to the buffer from the file.

Return

On exit:

- `R0` contains zero if the call is successful.
- If `R0` contains the same value as word 3, the call has failed and EOF is assumed.
- If `R0` contains a smaller value than word 3, the call was partially successful. No error is assumed, but the buffer has not been filled.

If the handle is for an interactive device, that is, `SYS_ISTTY` returns –1. A nonzero return from `SYS_READ` indicates that the line read did not fill the buffer.

11.18 SYS_READC (0x07)

Reads a byte from the console.

Entry

Register `R1` must contain zero. There are no other parameters or values possible.

Return

On exit, `R0` contains the byte read from the console.

11.19 SYS_REMOVE (0x0E)

Deletes a specified file on the host filing system.

Entry

On entry, `R1` contains a pointer to a two-word argument block:

word 1
points to a null-terminated string that gives the path name of the file to be deleted

word 2
the length of the string.

Return

On exit, `R0` contains:

- 0 if the delete is successful
- a nonzero, host-specific error code if the delete fails.

11.20 SYS_RENAME (0x0F)

Renames a specified file.

Entry

On entry, `R1` contains a pointer to a four-word data block:

word 1
a pointer to the name of the old file

word 2
the length of the old filename

word 3
a pointer to the new filename

word 4
the length of the new filename.

Both strings are null-terminated.

Return

On exit, `R0` contains:

- 0 if the rename is successful
- a nonzero, host-specific error code if the rename fails.

11.21 SYS_SEEK (0x0A)

Seeks to a specified position in a file using an offset specified from the start of the file.

The file is assumed to be a byte array and the offset is given in bytes.

Entry

On entry, `R1` contains a pointer to a two-word data block:

word 1
a handle for a seekable file object

word 2
the absolute byte position to search to.

Return

On exit, `R0` contains:

- 0 if the request is successful
- A negative value if the request is not successful. Use `SYS_ERRNO` to read the value of the host `errno` variable describing the error.

———— **Note** ————

The effect of seeking outside the current extent of the file object is undefined.

————————————

11.22 SYS_SYSTEM (0x12)

Passes a command to the host command-line interpreter.

This enables you to execute a system command such as `dir`, `ls`, or `pwd`. The terminal I/O is on the host, and is not visible to the target.

——— **Caution** ———

The command passed to the host is executed on the host. Ensure that any command passed has no unintended consequences.

———————————

Entry

On entry, `R1` contains a pointer to a two-word argument block:

word 1
points to a string to be passed to the host command-line interpreter

word 2
the length of the string.

Return

On exit, `R0` contains the return status.

11.23 SYS_TICKFREQ (0x31)

Returns the tick frequency.

Entry

Register `R1` must contain 0 on entry to this routine.

Return

On exit, `R0` contains either:

- the number of ticks per second
- –1 if the target does not know the value of one tick. Some debuggers might not support this SVC when connected though RVI or DSTREAM and they always return –1 in `R0`.

11.24 SYS_TIME (0x11)

Returns the number of seconds since 00:00 January 1, 1970.

This is real-world time, regardless of any debug agent configuration, such as RVI or DSTREAM.

Entry

There are no parameters.

Return

On exit, `R0` contains the number of seconds.

11.25 SYS_TMPNAM (0x0D)

Returns a temporary name for a file identified by a system file identifier.

Entry

On entry, `R1` contains a pointer to a three-word argument block:

word 1
A pointer to a buffer.

word 2
A target identifier for this filename. Its value must be an integer in the range 0 to 255.

word 3
Contains the length of the buffer. The length must be at least the value of `L_tmpnam` on the host system.

Return

On exit, `R0` contains:

- 0 if the call is successful
- –1 if an error occurs.

The buffer pointed to by `R1` contains the filename, prefixed with a suitable directory name.

If you use the same target identifier again, the same filename is returned.

Note

The returned string must be null-terminated.

11.26 SYS_WRITE (0x05)

Writes the contents of a buffer to a specified file at the current file position.

The file position is specified either:

- explicitly, by a `SYS_SEEK`
- implicitly as one byte beyond the previous `SYS_READ` or `SYS_WRITE` request.

The file position is at the start of the file when the file is opened, and is lost when the file is closed.

Perform the file operation as a single action whenever possible. For example, do not split a write of 16KB into four 4KB chunks unless there is no alternative.

Entry

On entry, `R1` contains a pointer to a three-word data block:

word 1
contains a handle for a file previously opened with `SYS_OPEN`

word 2
points to the memory containing the data to be written

word 3
contains the number of bytes to be written from the buffer to the file.

Return

On exit, `R0` contains:

- 0 if the call is successful
- the number of bytes that are not written, if there is an error.

11.27 SYS_WRITEC (0x03)

Writes a character byte, pointed to by `R1`, to the debug channel.

When executed under an ARM debugger, the character appears on the host debugger console.

Entry

On entry, `R1` contains a pointer to the character.

Return

None. Register `R0` is corrupted.

11.28 SYS_WRITE0 (0x04)

Writes a null-terminated string to the debug channel.

When executed under an ARM debugger, the characters appear on the host debugger console.

Entry

On entry, `R1` contains a pointer to the first byte of the string.

Return

None. Register `R0` is corrupted.

Chapter 12
ARMv6 SIMD Instruction Intrinsics

Describes the ARMv6 SIMD instruction intrinsics. SIMD instructions allow the processor to operate on packed 8-bit or 16-bit values in 32-bit registers.

It contains the following sections:

- *12.1 ARMv6 SIMD intrinsics by prefix* on page 12-760.
- *12.2 ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags* on page 12-762.
- *12.3 ARMv6 SIMD intrinsics, compatible processors and architectures* on page 12-765.
- *12.4 ARMv6 SIMD instruction intrinsics and APSR GE flags* on page 12-766.
- *12.5 __qadd16 intrinsic* on page 12-768.
- *12.6 __qadd8 intrinsic* on page 12-769.
- *12.7 __qasx intrinsic* on page 12-770.
- *12.8 __qsax intrinsic* on page 12-771.
- *12.9 __qsub16 intrinsic* on page 12-772.
- *12.10 __qsub8 intrinsic* on page 12-773.
- *12.11 __sadd16 intrinsic* on page 12-774.
- *12.12 __sadd8 intrinsic* on page 12-775.
- *12.13 __sasx intrinsic* on page 12-776.
- *12.14 __sel intrinsic* on page 12-777.
- *12.15 __shadd16 intrinsic* on page 12-778.
- *12.16 __shadd8 intrinsic* on page 12-779.
- *12.17 __shasx intrinsic* on page 12-780.
- *12.18 __shsax intrinsic* on page 12-781.
- *12.19 __shsub16 intrinsic* on page 12-782.
- *12.20 __shsub8 intrinsic* on page 12-783.
- *12.21 __smlad intrinsic* on page 12-784.
- *12.22 __smladx intrinsic* on page 12-785.

- *12.23 __smlald intrinsic* on page 12-786.
- *12.24 __smlaldx intrinsic* on page 12-787.
- *12.25 __smlsd intrinsic* on page 12-788.
- *12.26 __smlsdx intrinsic* on page 12-789.
- *12.27 __smlsld intrinsic* on page 12-790.
- *12.28 __smlsldx intrinsic* on page 12-791.
- *12.29 __smuad intrinsic* on page 12-792.
- *12.30 __smuadx intrinsic* on page 12-793.
- *12.31 __smusd intrinsic* on page 12-794.
- *12.32 __smusdx intrinsic* on page 12-795.
- *12.33 __ssat16 intrinsic* on page 12-796.
- *12.34 __ssax intrinsic* on page 12-797.
- *12.35 __ssub16 intrinsic* on page 12-798.
- *12.36 __ssub8 intrinsic* on page 12-799.
- *12.37 __sxtab16 intrinsic* on page 12-800.
- *12.38 __sxtb16 intrinsic* on page 12-801.
- *12.39 __uadd16 intrinsic* on page 12-802.
- *12.40 __uadd8 intrinsic* on page 12-803.
- *12.41 __uasx intrinsic* on page 12-804.
- *12.42 __uhadd16 intrinsic* on page 12-805.
- *12.43 __uhadd8 intrinsic* on page 12-806.
- *12.44 __uhasx intrinsic* on page 12-807.
- *12.45 __uhsax intrinsic* on page 12-808.
- *12.46 __uhsub16 intrinsic* on page 12-809.
- *12.47 __uhsub8 intrinsic* on page 12-810.
- *12.48 __uqadd16 intrinsic* on page 12-811.
- *12.49 __uqadd8 intrinsic* on page 12-812.
- *12.50 __uqasx intrinsic* on page 12-813.
- *12.51 __uqsax intrinsic* on page 12-814.
- *12.52 __uqsub16 intrinsic* on page 12-815.
- *12.53 __uqsub8 intrinsic* on page 12-816.
- *12.54 __usad8 intrinsic* on page 12-817.
- *12.55 __usada8 intrinsic* on page 12-818.
- *12.56 __usat16 intrinsic* on page 12-819.
- *12.57 __usax intrinsic* on page 12-820.
- *12.58 __usub16 intrinsic* on page 12-821.
- *12.59 __usub8 intrinsic* on page 12-822.
- *12.60 __uxtab16 intrinsic* on page 12-823.
- *12.61 __uxtb16 intrinsic* on page 12-824.

12.1 ARMv6 SIMD intrinsics by prefix

The following table shows the intrinsics according to prefix name.

Each intrinsic's prefix indicates the type of arithmetic performed, as follows:

- `__s`, signed.
- `__q`, signed saturating.
- `__sh`, signed halving.
- `__u`, unsigned.
- `__uq`, unsigned saturating.
- `__uh`, unsigned halving.

The `__sel()` intrinsic falls outside the classifications shown in the table. This intrinsic selects bytes according to GE bit values.

Table 12-1 ARMv6 SIMD intrinsics by prefix

	ARMv6 SIMD instruction intrinsics grouped by prefix					
Intrinsic classification	`__s`	`__q`	`__sh`	`__u`	`__uq`	`__uh`
Byte addition	`__sadd8`	`__qadd8`	`__shadd8`	`__uadd8`	`__uqadd8`	`__uhadd8`
Byte subtraction	`__ssub8`	`__qsub8`	`__shsub8`	`__usub8`	`__uqsub8`	`__uhsub8`
Halfword addition	`__sadd16`	`__qadd16`	`__shadd16`	`__uadd16`	`__uqadd16`	`__uhadd16`
Halfword subtraction	`__ssub16`	`__qsub16`	`__shsub16`	`__usub16`	`__uqsub16`	`__uhsub16`
Exchange halfwords within one operand, add high halfwords, subtract low halfwords	`__sasx`	`__qasx`	`__shasx`	`__uasx`	`__uqasx`	`__uhasx`
Exchange halfwords within one operand, subtract high halfwords, add low halfwords	`__ssax`	`__qsax`	`__shsax`	`__usax`	`__uqsax`	`__uhsax`
Unsigned sum of absolute difference	-	-	-	`__usad8`	-	-
Unsigned sum of absolute difference and accumulate	-	-	-	`__usada8`	-	-
Saturation to selected width	`__ssat16`	-	-	`__usat16`	-	-
Extract values (bit positions [23:16][7:0]), zero-extend to 16 bits	-	-	-	`__uxtb16`	-	-
Extract values (bit positions [23:16][7:0]) from second operand, zero-extend to 16 bits, add to first operand	-	-	-	`__uxtab16`	-	-
Sign-extend	`__sxtb16`	-	-	-	-	-
Sign-extend, add	`__sxtab16`	-	-	-	-	-
Signed multiply, add products	`__smuad`	-	-	-	-	-
Exchange halfwords of one operand, signed multiply, add products	`__smuadx`	-	-	-	-	-
Signed multiply, subtract products	`__smusd`	-	-	-	-	-
Exchange halfwords of one operand, signed multiply, subtract products	`__smusdx`	-	-	-	-	-
Signed multiply, add both results to another operand	`__smlad`	-	-	-	-	-

Table 12-1 ARMv6 SIMD intrinsics by prefix (continued)

	ARMv6 SIMD instruction intrinsics grouped by prefix					
Intrinsic classification	`__s`	`__q`	`__sh`	`__u`	`__uq`	`__uh`
Exchange halfwords of one operand, perform 2x16-bit multiplication, add both results to another operand	`__smladx`	-	-	-	-	-
Perform 2x16-bit multiplication, add both results to another operand	`__smlald`	-	-	-	-	-
Exchange halfwords of one operand, perform 2x16-bit multiplication, add both results to another operand	`__smlaldx`	-	-	-	-	-
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	`__smlsd`	-	-	-	-	-
Exchange halfwords of one operand, perform two signed 16-bit multiplications, add difference of products to a 32-bit accumulate operand	`__smlsdx`	-	-	-	-	-
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	`__smlsld`	-	-	-	-	-
Exchange halfwords of one operand, perform 2x16-bit multiplications, add difference of products to a 64-bit accumulate operand	`__smlsldx`	-	-	-	-	-

12.2 ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags

The following table describes each ARMv6 SIMD intrinsic, providing a summary description together with information about byte lanes and affected flags.

Table 12-2 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
`__sel`	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 `__sel` 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
`__smlad`	2x16-bit multiplication, adding both results to third operand.	int32	int16x2, int16x2, int32	Q bit
`__smladx`	Exchange halfwords of the second operand, 2x16-bit multiplication, adding both results to third operand.	int16x2	int16x2, int16x2	Q bit
`__smlald`	2x16-bit multiplication, adding both results to third operand. Overflow in addition is not detected.	int64	int16x2, int16x2, int64	None

Table 12-2 ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags (continued)

Intrinsic	Summary description	Byte lanes		Affected flags
		Returns	Operands	
`__smlaldx`	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
`__smlsd`	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
`__smlsdx`	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
`__smlsld`	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
`__smlsldx`	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
`__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

Table 12-2 ARMv6 SIMD intrinsics, summary descriptions, byte lanes, affected flags (continued)

Intrinsic	Summary description	Byte lanes		Affected flags
		Returns	Operands	
`__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
`__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

12.3 ARMv6 SIMD intrinsics, compatible processors and architectures

The following table 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 12-3 ARMv6 SIMD intrinsics, compatible processors and architectures

Intrinsics	Compatible `--cpu` options
`__qadd16`, `__qadd8`, `__qasx`	`6, 6K, 6T2, 6Z, 7-R, Cortex-R4, Cortex-R4F, Cortex-R7, Cortex-R7.no_vfp, Cortex-M4, Cortex-M4.fp.sp, Cortex-M7, Cortex-M7.fp.sp, Cortex-M7.fp.dp, ARM1136J-S, ARM1136JF-S, ARM1136J-S-rev1, ARM1136JF-S-rev1, ARM1156T2-S, ARM1156T2F-S, ARM1176JZ-S, ARM1176JZF-S, MPCore, MPCore.no_vfp, MPCoreNoVFP`

Related references

7.28 --cpu=list on page 7-301.

7.29 --cpu=name compiler option on page 7-302.

12.4 ARMv6 SIMD instruction intrinsics and APSR GE flags

The following table describes the action and operation of the APSR.GE flags for each ARMv6 SIMD instruction intrinsic.

Table 12-4 ARMv6 SIMD instruction intrinsics and APSR GE flags

Intrinsic	APSR.GE flag action	APSR.GE operation

`__sel`	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`

Table 12-4 ARMv6 SIMD instruction intrinsics and APSR GE flags (continued)

Intrinsic	APSR.GE flag action	APSR.GE operation

`__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`

12.5 __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 $-2^{15} \leq x \leq 2^{15} - 1$.

Syntax

`unsigned int __qadd16(unsigned int *val1*, unsigned int *val2*)`

Where:

val1
holds the first two 16-bit summands

val2
holds the second two 16-bit summands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __qadd16(val1, val2); /* res[15:0] = val1[15:0] + val2[15:0]
                                   res[16:31] = val1[31:16] + val2[31:16]
                                 */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

QADD16.
Saturating instructions.
ARM and Thumb instruction summary.

12.6 __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 $-2^7 \leq x \leq 2^7 - 1$.

Syntax

`unsigned int __qadd8(unsigned int val1, unsigned int val2)`

Where:

`val1`
holds the first four 8-bit summands

`val2`
holds the other four 8-bit summands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __qadd8(val1,val2); /* res[7:0] = val1[7:0] + val2[7:0]
                                 res[15:8] = val1[15:8] + val2[15:8]
                                 res[23:16] = val1[23:16] + val2[23:16]
                                 res[31:24] = val1[31:24] + val2[31:24]
                               */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

QADD8.

Saturating instructions.

ARM and Thumb instruction summary.

12.7 __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 $-2^{15} \leq x \leq 2^{15} - 1$.

Syntax

`unsigned int __qasx(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`

holds the first operand for the subtraction in the low halfword, and the first operand for the addition in the high halfword

`val2`

holds the second operand for the subtraction in the high halfword, and the second operand for the addition in the low halfword.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __qasx(val1,val2); /* res[15:0] = val1[15:0] - val2[31:16]
                                res[31:16] = val1[31:16] + val2[15:0]
                              */
                             /* Alternative equivalent representation:
                                val2[15:0][31:16] = val2[31:16][15:0]
                                res[15:0] = val1[15:0] - val2[15:0]
                                res[31:16] = val[31:16] + val2[31:16]
                              */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

QASX.

Saturating instructions.

ARM and Thumb instruction summary.

12.8 __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 $-2^{15} \leq x \leq 2^{15} - 1$.

Syntax

`unsigned int __qsax(unsigned int *val1*, unsigned int *val2*)`

Where:

val1
: holds the first operand for the addition in the low halfword, and the first operand for the subtraction in the high halfword

val2
: holds the second operand for the addition in the high halfword, and the second operand for the subtraction in the low halfword.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __qsax(val1,val2); /* res[15:0] = val1[15:0] + val2[31:16]
                                res[31:16] = val1[31:16] - val2[15:0]
                              */
                             /* Alternative equivalent representation:
                                val2[15:0][31:16] = val2[31:16][15:0]
                                res[15:0] = val1[15:0] + val2[15:0]
                                res[31:16] = val[31:16] - val2[31:16]
                              */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

QSAX.

Saturating instructions.

ARM and Thumb instruction summary.

12.9 __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 $-2^{15} \leq x \leq 2^{15} - 1$.

Syntax

`unsigned int __qsub16(unsigned int *val1*, unsigned int *val2*)`

Where:

`val1`
holds the first halfword operands

`val2`
holds the second halfword operands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __qsub16(val1,val2); /* res[15:0] = val1[15:0] - val2[15:0]
                                  res[31:16] = val1[31:16] - val2[31:16]
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

QSUB16.

Saturating instructions.

ARM and Thumb instruction summary.

12.10 __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 $-2^7 \leq x \leq 2^7 - 1$.

Syntax

`unsigned int __qsub8(unsigned int *val1*, unsigned int *val2*)`

Where:

`val1`

holds the first four 8-bit operands

`val2`

holds the second four 8-bit operands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __qsub8(val1,val2); /* res[7:0] = val1[7:0] - val2[7:0]
                                 res[15:8] = val1[15:8] - val2[15:8]
                                 res[23:16] = val1[23:16] - val2[23:16]
                                 res[31:24] = val1[31:24] - val2[31:24]
                               */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

QSUB8.

Saturating instructions.

ARM and Thumb instruction summary.

12.11 __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 *Application Program Status Register* (APSR) are set according to the results of the additions.

Syntax

`unsigned int __sadd16(unsigned int *val1*, unsigned int *val2*)`

Where:

`val1`
holds the first two 16-bit summands

`val2`
holds the second two 16-bit summands.

Return value

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] ≥ 0 then APSR.GE[1:0] = 11 else 00.
- If *`res`*[31:16] ≥ 0 then APSR.GE[3:2] = 11 else 00.

Example

```
unsigned int add_halfwords(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __sadd16(val1,val2); /* res[15:0] = val1[15:0] + val2[15:0]
                                  res[31:16] = val1[31:16] + val2[31:16]
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

12.14 __sel intrinsic on page 12-777.

Related information

SADD16.

Saturating instructions.

ARM and Thumb instruction summary.

12.12 __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.

Syntax

`unsigned int __sadd8(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first four 8-bit summands

`val2`
holds the second four 8-bit summands.

Return value

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] ≥ 0 then APSR.GE[0] = 1 else 0.
- If *`res`*[15:8] ≥ 0 then APSR.GE[1] = 1 else 0.
- If *`res`*[23:16] ≥ 0 then APSR.GE[2] = 1 else 0.
- If *`res`*[31:24] ≥ 0 then APSR.GE[3] = 1 else 0.

Example

```
unsigned int add_bytes(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __sadd16(val1,val2); /* res[7:0] = val1[7:0] + val2[7:0]
                                  res[15:8] = val1[15:8] + val2[15:8]
                                  res[23:16] = val1[23:16] + val2[23:16]
                                  res[31:24] = val1[31:24] + val2[31:24]
                                */
    return res;
}
```

Related references

12.14 __sel intrinsic on page 12-777.

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SADD8.

Saturating instructions.

ARM and Thumb instruction summary.

12.13 __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.

Syntax

`unsigned int __sasx(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`

holds the first operand for the subtraction in the low halfword, and the first operand for the addition in the high halfword

`val2`

holds the second operand for the subtraction in the high halfword, and the second operand for the addition in the low halfword.

Return value

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] ≥ 0 then APSR.GE[1:0] = 11 else 00.
- If *`res`*[31:16] ≥ 0 then APSR.GE[3:2] = 11 else 00.

Example

```
unsigned int exchange_subtract_add(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __sasx(val1,val2); /* res[15:0] = val1[15:0] - val2[31:16]
                                res[31:16] = val1[31:16] + val2[15:0]
                              */
    return res;
}
```

Related references

12.14 __sel intrinsic on page 12-777.

9.150 ARMv6 SIMD intrinsics on page 9-678.

12.14 __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 on 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 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 `__sasx` intrinsic.

Syntax

`unsigned int __sel(unsigned int val1, unsigned int val2)`

Where:

`val1`
holds four selectable bytes

`val2`
holds four selectable bytes.

Return value

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] = val1[7:0] else res[7:0] = val2[7:0]
if APSR.GE[1] == 1 then res[15:8] = val1[15:8] else res[15:8] = val2[15:8]
if APSR.GE[2] == 1 then res[23:16] = val1[23:16] else res[23:16] = val2[23:16]
if APSR.GE[3] == 1 then res[31:24] = val1[31:24] else res = val2[31:24]
```

Example

```
unsigned int ge_filter(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __sel(val1,val2);
    return res;
}
unsigned int foo(unsigned int a, unsigned int b)
{
    int res;
    int filtered_res;
    res = __sasx(a,b);  /* This intrinsic sets the GE flags */
    filtered_res = ge_filter(res); /* Filter the results of the __sasx */
                                   /* intrinsic. Some results are filtered */
                                   /* out based on the GE flags. */
    return filtered_res;
}
```

Related references

12.11 __sadd16 intrinsic on page 12-774.
12.13 __sasx intrinsic on page 12-776.
12.34 __ssax intrinsic on page 12-797.
12.36 __ssub8 intrinsic on page 12-799.
12.35 __ssub16 intrinsic on page 12-798.
9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SEL.
ARM and Thumb instruction summary.

12.15 __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.

Syntax

`unsigned int __shadd16(unsigned int *val1*, unsigned int *val2*)`

Where:

`val1`
holds the first two 16-bit summands

`val2`
holds the second two 16-bit summands.

Return value

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_halve(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __shadd16(val1,val2); /* res[15:0] = (val1[15:0] + val2[15:0]) >> 1
                                   res[31:16] = (val1[31:16] + val2[31:16]) >> 1
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SHADD16.

ARM and Thumb instruction summary.

12.16 __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.

Syntax

`unsigned int __shadd8(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first four 8-bit summands

`val2`
holds the second four 8-bit summands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __shadd8(val1,val2); /* res[7:0] = (val1[7:0] + val2[7:0]) >> 1
                                  res[15:8] = (val1[15:8] + val2[15:8]) >> 1
                                  res[23:16] = (val1[23:16] + val2[23:16]) >> 1
                                  res[31:24] = (val1[31:24] + val2[31:24]) >> 1
                               */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SHADD8.

ARM and Thumb instruction summary.

12.17 __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.

Syntax

`unsigned int __shasx(unsigned int val1, unsigned int val2)`

Where:

`val1`
: holds the first halfword operands

`val2`
: holds the second halfword operands.

Return value

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_halve(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __shasx(val1,val2); /* res[15:0] = (val1[15:0] - val2[31:16]) >> 1
                                 res[31:16] = (val1[31:16] - val2[15:0]) >> 1
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SHASX.

ARM and Thumb instruction summary.

12.18 __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.

Syntax

`unsigned int __shsax(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
: holds the first halfword operands

`val2`
: holds the second halfword operands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __shsax(val1,val2); /* res[15:0] = (val1[15:0] + val2[31:16]) >> 1
                                 res[31:16] = (val1[31:16] - val2[15:0]) >> 1
                               */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SHSAX.

ARM and Thumb instruction summary.

12.19 __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.

Syntax

`unsigned int __shsub16(unsigned int *val1*, unsigned int *val2*)`

Where:

`val1`
holds the first halfword operands

`val2`
holds the second halfword operands.

Return value

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_halve(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __shsub16(val1,val2); /* res[15:0] = (val1[15:0] - val2[15:0]) >> 1
                                   res[31:16] = (val1[31:16] - val2[31:16]) >> 1
                                 */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SHSUB16.

ARM and Thumb instruction summary.

12.20 __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.

Syntax

`unsigned int __shsub8(unsigned int *val1*, unsigned int *val2*)`

Where:

val1
holds the first four operands

val2
holds the second four operands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __shsub8(val1,val2); /* res[7:0] = (val1[7:0] - val2[7:0]) >> 1
                                  res[15:8] = (val1[15:8] - val2[15:8]) >> 1
                                  res[23:16] = (val1[23:16] - val2[23:16] >> 1
                                  res[31:24] = (val1[31:24] - val2[31:24] >> 1
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SHSUB8.

ARM and Thumb instruction summary.

12.21 __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.

Syntax

`unsigned int __smlad(unsigned int` *`val1`*`, unsigned int` *`val2`*`, unsigned int` *`val3`*`)`

Where:

`val1`
: holds the first halfword operands for each multiplication

`val2`
: holds the second halfword operands for each multiplication

`val3`
: holds the accumulate value.

Return 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 val1, unsigned int val2, unsigned int 
val3)
{
    unsigned int res;
    res = __smlad(val1,val2,val3); /* p1 = val1[15:0] × val2[15:0]
                                      p2 = val1[31:16] × val2[31:16]
                                      res[31:0] = p1 + p2 + val3[31:0]
                                    */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMLAD.
ARM and Thumb instruction summary.

12.22 __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.

Syntax

`unsigned int __smladx(unsigned int *val1*, unsigned int *val2*, unsigned int *val3*)`

Where:

`val1`
: holds the first halfword operands for each multiplication

`val2`
: holds the second halfword operands for each multiplication

`val3`
: holds the accumulate value.

Return 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 val1, unsigned int val2, unsigned int
val3)
{
    unsigned int res;
    res = __smladx(val1,val2,val3); /* p1 = val1[15:0] × val2[31:16]
                                       p2 = val1[31:16] × val2[15:0]
                                       res[31:0] = p1 + p2 + val3[31:0]
                                     */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMLAD.

ARM and Thumb instruction summary.

12.23 __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}.

Syntax

`unsigned long long __smlald(unsigned int *val1*, unsigned int *val2*, unsigned long long *val3*)`

Where:

val1
: holds the first halfword operands for each multiplication

val2
: holds the second halfword operands for each multiplication

val3
: holds the accumulate value.

Return value

The `__smlald` intrinsic returns the product of each multiplication added to the accumulate value.

Example

```
unsigned int dual_multiply_accumulate(unsigned int val1, unsigned int val2, unsigned int
val3)
{
    unsigned int res;
    res = __smlald(val1,val2,val3); /* p1 = val1[15:0] × val2[15:0]
                                       p2 = val1[31:16] × val2[31:16]
                                       sum = p1 + p2 + val3[63:32][31:0]
                                       res[63:32] = sum[63:32]
                                       res[31:0] = sum[31:0]
                                     */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMLALD.

ARM and Thumb instruction summary.

12.24 __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}.

Syntax

`unsigned long long __smlaldx(unsigned int` *`val1`*`, unsigned int` *`val2`*`, unsigned long long` *`val3`*`)`

Where:

`val1`
: holds the first halfword operands for each multiplication

`val2`
: holds the second halfword operands for each multiplication

`val3`
: holds the accumulate value.

Return value

The `__smlald` intrinsic returns the product of each multiplication added to the accumulate value.

Example

```
unsigned int dual_multiply_accumulate(unsigned int val1, unsigned int val2, unsigned int
val3)
{
    unsigned int res;
    res = __smlald(val1,val2,val3); /* p1 = val1[15:0] × val2[31:16]
                                       p2 = val1[31:16] × val2[15:0]
                                       sum = p1 + p2 + val3[63:32][31:0]
                                       res[63:32] = sum[63:32]
                                       res[31:0] = sum[31:0]
                                     */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMLALDX.

ARM and Thumb instruction summary.

12.25 __smlsd 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.

Syntax

`unsigned int __smlsd(unsigned int val1, unsigned int val2, unsigned int val3)`

Where:

`val1`
: holds the first halfword operands for each multiplication

`val2`
: holds the second halfword operands for each multiplication

`val3`
: holds the accumulate value.

Return 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 val1, unsigned int val2, unsigned int
val3)
{
    unsigned int res;
    res = __smlsd(val1,val2,val3); /* p1 = val1[15:0] × val2[15:0]
                                      p2 = val1[31:16] × val2[31:16]
                                      res[31:0] = p1 - p2 + val3[31:0]
                                    */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMLSD.

ARM and Thumb instruction summary.

12.26 __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.

Syntax

`unsigned int __smlsdx(unsigned int` *`val1`*`, unsigned int` *`val2`*`, unsigned int` *`val3`*`)`

Where:

`val1`
: holds the first halfword operands for each multiplication

`val2`
: holds the second halfword operands for each multiplication

`val3`
: holds the accumulate value.

Return 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 val1, unsigned int val2, unsigned int
val3)
{
    unsigned int res;
    res = __smlsd(val1,val2,val3); /* p1 = val1[15:0] × val2[31:16]
                                      p2 = val1[31:16] × val2[15:0]
                                      res[31:0] = p1 - p2 + val3[31:0]
                                    */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMLSDX.

ARM and Thumb instruction summary.

12.27 __smlsld intrinsic

This intrinsic inserts an `SMLSLD` 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}.

Syntax

`unsigned long long __smlsld(unsigned int val1, unsigned int val2, unsigned long long val3)`

Where:

val1
: holds the first halfword operands for each multiplication

val2
: holds the second halfword operands for each multiplication

val3
: holds the accumulate value.

Return value

The `__smlsld` 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 val1, unsigned int val2, unsigned
long long val3)
{
    unsigned int res;
    res = __smlsld(val1,val2,val3); /* p1 = val1[15:0] × val2[15:0]
                                       p2 = val1[31:16] × val2[31:16]
                                       res[63:0] = p1 - p2 + val3[63:0]
                                     */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMLSLD.

ARM and Thumb instruction summary.

12.28 __smlsldx intrinsic

This intrinsic inserts an `SMLSLDX` 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}.

Syntax

`unsigned long long __smlsldx(unsigned int val1, unsigned int val2, unsigned long long val3)`

Where:

val1	holds the first halfword operands for each multiplication
val2	holds the second halfword operands for each multiplication
val3	holds the accumulate value.

Return value

The `__smlsld` 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 val1, unsigned int val2, unsigned
long long val3)
{
    unsigned int res;
    res = __smlsld(val1,val2,val3); /* p1 = val1[15:0] × val2[31:16]
                                       p2 = val1[31:16] × val2[15:0]
                                       res[63:0] = p1 - p2 + val3[63:0]
                                     */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMLSLDX.

ARM and Thumb instruction summary.

12.29 __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.

Syntax

`unsigned int __smuad(unsigned int *val1*, unsigned int *val2*)`

Where:

val1
: holds the first halfword operands for each multiplication

val2
: holds the second halfword operands for each multiplication.

Return value

The `__smuad` intrinsic returns the products of the two 16-bit signed multiplications.

Example

```
unsigned int dual_multiply_prods(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __smuad(val1,val2); /* p1 = val1[15:0] × val2[15:0]
                                 p2 = val1[31:16] × val2[31:16]
                                 res[31:0] = p1 + p2
                               */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMUAD.

ARM and Thumb instruction summary.

12.30 __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 × bottom and bottom × top multiplication. The Q flag is set if the addition overflows. The multiplications cannot overflow.

Syntax

`unsigned int __smuadx(unsigned int *val1*, unsigned int *val2*)`

Where:

val1
: holds the first halfword operands for each multiplication

val2
: holds the second halfword operands for each multiplication.

Return value

The `__smuadx` intrinsic returns the products of the two 16-bit signed multiplications.

Example

```
unsigned int exchange_dual_multiply_prods(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __smuadx(val1,val2); /* val2[31:16][15:0] = val2[15:0][31:16]
                                  p1 = val1[15:0] × val2[15:0]
                                  p2 = val1[31:16] × val2[31:16]
                                  res[31:0] = p1 + p2
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMUADX.

ARM and Thumb instruction summary.

12.31 __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.

Syntax

`unsigned int __smusd(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first halfword operands for each multiplication

`val2`
holds the second halfword operands for each multiplication.

Return value

The `__smusd` intrinsic returns the difference of the products of the two 16-bit signed multiplications.

Example

```
unsigned int dual_multiply_prods(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __smuad(val1,val2); /* p1 = val1[15:0] × val2[15:0]
                                 p2 = val1[31:16] × val2[31:16]
                                 res[31:0] = p1 - p2
                               */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMUSD.

ARM and Thumb instruction summary.

12.32 __smusdx intrinsic

This intrinsic inserts an `SMUSDX` 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 × bottom and bottom × top multiplication.

Syntax

`unsigned int __smusdx(unsigned int *val1*, unsigned int *val2*)`

Where:

val1
: holds the first halfword operands for each multiplication

val2
: holds the second halfword operands for each multiplication.

Return value

The `__smusdx` intrinsic returns the difference of the products of the two 16-bit signed multiplications.

Example

```
unsigned int dual_multiply_prods(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __smuad(val1,val2); /* p1 = val1[15:0] × val2[31:16]
                                 p2 = val1[31:16] × val2[15:0]
                                 res[31:0] = p1 - p2
                               */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SMUSDX.

ARM and Thumb instruction summary.

12.33 __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.

Syntax

`unsigned int __saturate_halfwords(unsigned int val1, unsigned int val2)`

Where:

val1
holds the two signed 16-bit values to be saturated

val2
is the bit position for saturation, an integral constant expression in the range 1 to 16.

Return value

The `__ssat16` intrinsic returns:

- The signed saturation of the low halfword in *val1*, saturated to the bit position specified in *val2* and returned in the low halfword of the return value.
- The signed saturation of the high halfword in *val1*, saturated to the bit position specified in *val2* and returned in the high halfword of the return value.

Example

```
unsigned int saturate_halfwords(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __ssat16(val1,val2); /* Saturate halfwords in val1 to the signed
                                  range specified by the bit position in val2 */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SSAT16.

Saturating instructions.

ARM and Thumb instruction summary.

12.34 __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.

Syntax

`unsigned int __ssax(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
: holds the first operand for the addition in the low halfword, and the first operand for the subtraction in the high halfword

`val2`
: holds the second operand for the addition in the high halfword, and the second operand for the subtraction in the low halfword.

Return value

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] ≥ 0 then APSR.GE[1:0] = 11 else 00.
- If *`res`*[31:16] ≥ 0 then APSR.GE[3:2] = 11 else 00.

Example

```
unsigned int exchange_subtract_add(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __ssax(val1,val2); /* res[15:0] = val1[15:0] + val2[31:16]
                                res[31:16] = val1[31:16] - val2[15:0]
                              */
    return res;
}
```

Related references

12.14 __sel intrinsic on page 12-777.

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SSAX.

ARM and Thumb instruction summary.

12.35 __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.

Syntax

`unsigned int __ssub16(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first operands of each subtraction in the low and the high halfwords

`val2`
holds the second operands for each subtraction in the low and the high halfwords.

Return value

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] ≥ 0 then APSR.GE[1:0] = 11 else 00.
- If *`res`*[31:16] ≥ 0 then APSR.GE[3:2] = 11 else 00.

Example

```
unsigned int subtract halfwords(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __ssub16(val1,val2); /* res[15:0] = val1[15:0] - val2[15:0]
                                  res[31:16] = val1[31:16] - val2[31:16]
                                */
    return res;
}
```

Related references

12.14 __sel intrinsic on page 12-777.

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SSUB16.

ARM and Thumb instruction summary.

12.36 __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.

Syntax

`unsigned int __ssub8(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first four 8-bit operands of each subtraction

`val2`
holds the second four 8-bit operands of each subtraction.

Return value

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] ≥ 0 then APSR.GE[0] = 1 else 0.
- If *`res`*[15:8] ≥ 0 then APSR.GE[1] = 1 else 0.
- If *`res`*[23:16] ≥ 0 then APSR.GE[2] = 1 else 0.
- If *`res`*[31:24] ≥ 0 then APSR.GE[3] = 1 else 0.

Example

```
unsigned int subtract bytes(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __ssub8(val1,val2); /* res[7:0] = val1[7:0] - val2[7:0]
                                 res[15:8] = val1[15:8] - val2[15:8]
                                 res[23:16] = val1[23:16] - val2[23:16]
                                 res[31:24] = val1[31:24] - val2[31:24]
                               */
    return res;
}
```

Related references

12.14 __sel intrinsic on page 12-777.

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SSUB8.

ARM and Thumb instruction summary.

12.37 __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.

Syntax

`unsigned int __sxtab16(unsigned int *val1*, unsigned int *val2*)`

Where:

`val1`
: holds the values that the extracted and sign-extended values are added to

`val2`
: holds the two 8-bit values to be extracted and sign-extended.

Return value

The `__sxtab16` intrinsic returns the addition of *`val1`* and *`val2`*, where the 8-bit values in *`val2`*[7:0] and *`val2`*[23:16] have been extracted and sign-extended before the addition.

Example

```
unsigned int extract_sign_extend_and_add(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __sxtab16(val1,val2); /* res[15:0]
                                    = val1[15:0] + SignExtended(val2[7:0])
                                   res[31:16]
                                    = val1[31:16] + SignExtended(val2[23:16])
                                 */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SXTAB16.

ARM and Thumb instruction summary.

12.38 __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.

Syntax

`unsigned int __sxtb16(unsigned int *val*)`

Where *`val`*[7:0] and *`val`*[23:16] hold the two 8-bit values to be sign-extended.

Return value

The `__sxtb16` intrinsic returns the 8-bit values sign-extended to 16-bit values.

Example

```
unsigned int sign_extend(unsigned int val)
{
    unsigned int res;
    res = __sxtb16(val1,val2); /* res[15:0] = SignExtended(val[7:0]
                                  res[31:16] = SignExtended(val[23:16]
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

SXTB16.

ARM and Thumb instruction summary.

12.39 __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.

Syntax

`unsigned int __uadd16(unsigned int *val1*, unsigned int *val2*)`

Where:

`val1`	holds the first two halfword summands for each addition
`val2`	holds the second two halfword summands for each addition.

Return value

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] ≥ 0x10000 then APSR.GE[0] = 11 else 00.
- If *`res`*[31:16] ≥ 0x10000 then APSR.GE[1] = 11 else 00.

Example

```
unsigned int add_halfwords(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __uadd16(val1,val2); /* res[15:0] = val1[15:0] + val2[15:0]
                                  res[31:16] = val1[31:16] + val2[31:16]
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UADD16.

ARM and Thumb instruction summary.

12.40 __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.

Syntax

`unsigned int __uadd8(unsigned int *val1*, unsigned int *val2*)`

Where:

val1
holds the first four 8-bit summands for each addition

val2
holds the second four 8-bit summands for each addition.

Return value

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] ≥ 0x100 then APSR.GE[0] = 1 else 0.
- If *res*[15:8] ≥ 0x100 then APSR.GE[1] = 1 else 0.
- If *res*[23:16] ≥ 0x100 then APSR.GE[2] = 1 else 0.
- If *res*[31:24] ≥ 0x100 then APSR.GE[3] = 1 else 0.

Example

```
unsigned int add_bytes(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __uadd8(val1,val2); /* res[7:0] = val1[7:0] + val2[7:0]
                                 res[15:8] = val1[15:8] + val2[15:8]
                                 res[23:16] = val1[23:16] + val2[23:16]
                                 res[31:24] = val1[31:24] + val2[31:24]
                               */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UADD8.

ARM and Thumb instruction summary.

12.41 __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.

Syntax

`unsigned int __uasx(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
: holds the first operand for the subtraction in the low halfword, and the first operand for the addition in the high halfword

`val2`
: holds the second operand for the subtraction in the high halfword and the second operand for the addition in the low halfword.

Return value

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] ≥ 0 then APSR.GE[1:0] = 11 else 00.
- If *`res`*[31:16] ≥ 0x10000 then APSR.GE[3:2] = 11 else 00.

Example

```
unsigned int exchange_add_subtract(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __uasx(val1,val2); /* res[15:0] = val1[15:0] - val2[31:16]
                                res[31:16] = val1[31:16] + val2[15:0]
                              */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UASX.

ARM and Thumb instruction summary.

12.42 __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.

Syntax

`unsigned int __uhadd16(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first two 16-bit summands

`val2`
holds the second two 16-bit summands.

Return value

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_halve(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __uhadd16(val1,val2); /* res[15:0] = (val1[15:0] + val2[15:0]) >> 1
                                   res[31:16] = (val1[31:16] + val2[31:16]) >> 1
                                 */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UHADD16.

ARM and Thumb instruction summary.

12.43 __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.

Syntax

`unsigned int __uhadd8(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first four 8-bit summands

`val2`
holds the second four 8-bit summands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __uhadd8(val1,val2); /* res[7:0] = (val1[7:0] + val2[7:0]) >> 1
                                  res[15:8] = (val1[15:8] + val2[15:8]) >> 1
                                  res[23:16] = (val1[23:16] + val2[23:16]) >> 1
                                  res[31:24] = (val1[31:24] + val2[31:24]) >> 1
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UHADD8.

ARM and Thumb instruction summary.

12.44 __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.

Syntax

`unsigned int __uhasx(unsigned int *val1*, unsigned int *val2*)`

Where:

`val1`
: holds the first operand for the subtraction in the low halfword, and the first operand for the addition in the high halfword

`val2`
: holds the second operand for the subtraction in the high halfword, and the second operand for the addition in the low halfword.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __uhasx(val1,val2); /* res[15:0] = (val1[15:0] - val2[31:16]) >> 1
                                 res[31:16] = (val1[31:16] + val2[15:0]) >> 1
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UHASX.

ARM and Thumb instruction summary.

12.45 __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.

Syntax

`unsigned int __uhsax(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first operand for the addition in the low halfword, and the first operand for the subtraction in the high halfword

`val2`
holds the second operand for the addition in the high halfword, and the second operand for the subtraction in the low halfword.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __uhsax(val1,val2); /* res[15:0] = (val1[15:0] + val2[31:16]) >> 1
                                 res[31:16] = (val1[31:16] - val2[15:0]) >> 1
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UHSAX.

ARM and Thumb instruction summary.

12.46 __uhsub16 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.

Syntax

`unsigned int __uhsub16(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
: holds the first two 16-bit operands

`val2`
: holds the second two 16-bit operands.

Return value

The `__uhsub16` 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 val1, unsigned int val2)
{
    unsigned int res;
    res = __uhsub16(val1,val2); /* res[15:0] = (val1[15:0] + val2[15:0]) >> 1
                                   res[31:16] = (val1[31:16] - val2[31:16]) >> 1
                                 */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UHSUB16.

ARM and Thumb instruction summary.

12.47 __uhsub8 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.

Syntax

`unsigned int __uhsub8(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`

holds the first four 8-bit operands

`val2`

holds the second four 8-bit operands.

Return value

The `__uhsub8` 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 val1, unsigned int val2)
{
    unsigned int res;
    res = __uhsub8(val1,val2); /* res[7:0] = (val1[7:0] - val2[7:0]) >> 1
                                  res[15:8] = (val1[15:8] - val2[15:8]) >> 1
                                  res[23:16] = (val1[23:16] - val2[23:16]) >> 1
                                  res[31:24] = (val1[31:24] - val2[31:24]) >> 1
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

12.48 __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 $0 \leq x \leq 2^{16} - 1$.

Syntax

`unsigned int __uqadd16(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first two halfword summands

`val2`
holds the second two halfword summands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __uqadd16(val1,val2); /* res[15:0] = val1[15:0] + val2[15:0]
                                   res[31:16] = val1[31:16] + val2[31:16]
                                 */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UQADD16.

ARM and Thumb instruction summary.

12.49 __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 $0 \leq x \leq 2^8 - 1$.

Syntax

`unsigned int __uqadd8(unsigned int val1, unsigned int val2)`

Where:

val1
holds the first four 8-bit summands

val2
holds the second four 8-bit summands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __uqadd8(val1,val2); /* res[7:0] = val1[7:0] + val2[7:0]
                                  res[15:8] = val1[15:8] + val2[15:8]
                                  res[23:16] = val1[23:16] + val2[23:16]
                                  res[31:24] = val1[31:24] + val2[31:24]
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UQADD8.

ARM and Thumb instruction summary.

12.50 __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 $0 \leq x \leq 2^{16} - 1$.

Syntax

`unsigned int __uqasx(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first two halfword operands

`val2`
holds the second two halfword operands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __uqasx(val1,val2); /* res[15:0] = val1[15:0] - val2[31:16]
                                 res[31:16] = val1[31:16] + val2[15:0]
                               */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UQASX.

ARM and Thumb instruction summary.

12.51 __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 $0 \leq x \leq 2^{16} - 1$.

Syntax

`unsigned int __uqsax(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`

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

`val2`

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.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __uqsax(val1,val2); /* res[15:0] = val1[15:0] + val2[31:16]
                                 res[31:16] = val1[31:16] - val2[15:0]
                               */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UQSAX.

ARM and Thumb instruction summary.

12.52 __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 $0 \leq x \leq 2^{16} - 1$.

Syntax

`unsigned int __uqsub16(unsigned int val1, unsigned int val2)`

Where:

val1	holds the first halfword operands for each subtraction
val2	holds the second halfword operands for each subtraction.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __uqsub16(val1,val2); /* res[15:0] = val1[15:0] - val2[15:0]
                                   res[31:16] = val1[31:16] - val2[31:16]
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UQSUB16.

ARM and Thumb instruction summary.

12.53 __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 $0 \leq x \leq 2^8 - 1$.

Syntax

`unsigned int __uqsub8(unsigned int *val1*, unsigned int *val2*)`

Where:

val1
holds the first four 8-bit operands

val2
holds the second four 8-bit operands.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __uqsub8(val1,val2); /* res[7:0] = val1[7:0] - val2[7:0]
                                  res[15:8] = val1[15:8] - val2[15:8]
                                  res[23:16] = val1[23:16] - val2[23:16]
                                  res[31:24] = val1[31:24] - val2[31:24]
                               */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UQSUB8.

ARM and Thumb instruction summary.

12.54 __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.

Syntax

`unsigned int __usad8(unsigned int val1, unsigned int val2)`

Where:

val1
holds the first four 8-bit operands for the subtractions

val2
holds the second four 8-bit operands for the subtractions.

Return value

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 val1, unsigned int val2)
{
    unsigned int res;
    res = __usad8(val1,val2); /* absdiff1 = val1[7:0] - val2[7:0]
                                 absdiff2 = val1[15:8] - val2[15:8]
                                 absdiff3 = val1[23:16] - val2[23:16]
                                 absdiff4 = val1[31:24] - val2[31:24]
                                 res[31:0] = absdiff1 + absdiff2 + absdiff3
                                  + absdiff4
                              */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

USAD8.

ARM and Thumb instruction summary.

12.55 __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.

Syntax

`unsigned int __usada8(unsigned int *val1*, unsigned int *val2*, unsigned int *val3*)`

Where:

val1
holds the first four 8-bit operands for the subtractions

val2
holds the second four 8-bit operands for the subtractions

val3
holds the accumulation value.

Return 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 val1, unsigned int val2, unsigned int
val3)
{
    unsigned int res;
    res = __usada8(val1,val2,val3); /* absdiff1 = val1[7:0] - val2[7:0]
                                       absdiff2 = val1[15:8] - val2[15:8]
                                       absdiff3 = val1[23:16] - val2[23:16]
                                       absdiff4 = val1[31:24] - val2[31:24]
                                       sum = absdiff1 + absdiff2 + absdiff3
                                        + absdiff4
                                       res[31:0] = sum[31:0] + val3[31:0]
                              */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

USADA8.

ARM and Thumb instruction summary.

12.56 __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.

Syntax

`unsigned int __usat16(unsigned int val1, /* constant */ unsigned int val2)`

Where:

val1
: holds the two 16-bit values that are to be saturated

val2
: specifies the bit position for saturation, and must be an integral constant expression.

Return value

The `__usat16` intrinsic returns the saturation of the two signed 16-bit values, as non-negative values.

Example

```
unsigned int saturate_halfwords(unsigned int val1)
{
    unsigned int res;
    #define VAL2 12
    res = __usat16(val1,VAL2); /* Saturate halfwords in val1 to the unsigned
                                   range specified by the bit position in VAL2
                                 */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

USAT16.

ARM and Thumb instruction summary.

12.57 __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.

Syntax

`unsigned int __usax(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first operand for the addition in the low halfword, and the first operand for the subtraction in the high halfword

`val2`
holds the second operand for the addition in the high halfword, and the second operand for the subtraction in the low halfword.

Return value

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] ≥ 0x10000 then APSR.GE[1:0] = 11 else 00.
- If *`res`*[31:16] ≥ 0 then APSR.GE[3:2] = 11 else 00.

Example

```
unsigned int exchange_subtract_add(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __usax(val1,val2); /* res[15:0] = val1[15:0] + val2[31:16]
                                res[31:16] = val1[31:16] - val2[15:0]
                              */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

USAX.

ARM and Thumb instruction summary.

12.58 __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.

Syntax

`unsigned int __usub16(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where:

`val1`
holds the first two halfword operands

`val2`
holds the second two halfword operands.

Return value

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] ≥ 0 then APSR.GE[1:0] = 11 else 00.
- If *`res`*[31:16] ≥ 0 then APSR.GE[3:2] = 11 else 00.

Example

```
unsigned int subtract_halfwords(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __usub16(val1,val2); /* res[15:0] = val1[15:0] - val2[15:0]
                                  res[31:16] = val1[31:16] - val2[31:16]
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

USUB16.

ARM and Thumb instruction summary.

12.59 __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.

Syntax

`unsigned int __usub8(unsigned int *val1*, unsigned int *val2*)`

Where:

`val1`
: holds the first four 8-bit operands

`val2`
: holds the second four 8-bit operands.

Return value

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] ≥ 0 then APSR.GE[0] = 1 else 0.
- If *`res`*[15:8] ≥ 0 then APSR.GE[1] = 1 else 0.
- If *`res`*[23:16] ≥ 0 then APSR.GE[2] = 1 else 0.
- If *`res`*[31:24] ≥ 0 then APSR.GE[3] = 1 else 0.

Example

```
unsigned int subtract(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __usub8(val1,val2);  /* res[7:0] = val1[7:0] - val2[7:0]
                                  res[15:8] = val1[15:8] - val2[15:8]
                                  res[23:16] = val1[23:16] - val2[23:16]
                                  res[31:24] = val1[31:24] - val2[31:24]
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

USUB8.

ARM and Thumb instruction summary.

12.60 __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.

Syntax

`unsigned int __uxtab16(unsigned int` *`val1`*`, unsigned int` *`val2`*`)`

Where *`val2`*[7:0] and *`val2`*[23:16] hold the two 8-bit values to be zero-extended.

Return value

The `__uxtab16` intrinsic returns the 8-bit values in *`val2`*, zero-extended to 16-bit values and added to *`val1`*.

Example

```
unsigned int extend_add(unsigned int val1, unsigned int val2)
{
    unsigned int res;
    res = __uxtab16(val1,val2); /* res[15:0] = ZeroExt(val2[7:0] to 16 bits)
                                     + val1[15:0]
                                   res[31:16] = ZeroExt(val2[31:16] to 16 bits)
                                     + val1[31:16]
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UXTAB16.

ARM and Thumb instruction summary.

12.61 __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.

Syntax

`unsigned int __uxtb16(unsigned int` *`val`*`)`

Where *`val`*[7:0] and *`val`*[23:16] hold the two 8-bit values to be zero-extended.

Return value

The `__uxtb16` intrinsic returns the 8-bit values zero-extended to 16-bit values.

Example

```
unsigned int zero_extend(unsigned int val)
{
    unsigned int res;
    res = __uxtb16(val1,val2); /* res[15:0] = ZeroExtended(val[7:0])
                                  res[31:16] = ZeroExtended(val[23:16])
                                */
    return res;
}
```

Related references

9.150 ARMv6 SIMD intrinsics on page 9-678.

Related information

UXTB16.

ARM and Thumb instruction summary.

Chapter 13
Via File Syntax

Describes the syntax of via files accepted by the `armcc`.

It contains the following sections:

- *13.1 Overview of via files* on page 13-826.
- *13.2 Via file syntax rules* on page 13-827.

13.1 Overview of via files

Via files are plain text files that allow you to specify compiler command-line arguments and options.

Typically, you use a via file to overcome the command-line length limitations. However, you might want to create multiple via files that:
- Group similar arguments and options together.
- Contain different sets of arguments and options to be used in different scenarios.

———— **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.

————————————

Via file evaluation

When the compiler 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.

Related references

13.2 Via file syntax rules on page 13-827.

7.170 --via=filename on page 7-455.

13.2 Via file syntax rules

Via files must conform to some 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
  ```

  This 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.

  Use quotation marks 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)

  Use apostrophes to delimit words that contain quotes, for example:

  `-DNAME='"ARM Compiler"'` (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"`

  This 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. A semicolon or hash character that appears anywhere else in a line 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 `'-D`*`symbol`*`="`*`value`*`"'` or as `-Dsymbol='"value"'`. For example:

  ```
  -c -DFOO_VALUE='"FOO_VALUE"'
  ```

Related concepts

13.1 Overview of via files on page 13-826.

Related references

7.170 --via=filename on page 7-455.

Chapter 14
Summary Table of GNU Language Extensions

Describes ARM compiler support for GNU extensions to the C and C++ languages.

It contains the following sections:

- *14.1 Supported GNU extensions* on page 14-829.

14.1 Supported GNU extensions

Describes ARM compiler support for GNU extensions to the C and C++ languages.

Table 14-1 Supported GNU extensions

GNU extension	Origin	Modes supported
9.4 __alignof__ on page 9-518	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++.
asm 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.
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 [g]	Standard C++.	C++, GNU C90, GNU C99, GNU C++.
9.28 Function attributes on page 9-545	-	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.157 GNU built-in functions on page 9-689	-	-
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++.
9.29 __attribute__((alias)) function attribute on page 9-547	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.57 __attribute__((aligned)) type attribute on page 9-576	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.63 __attribute__((aligned)) variable attribute on page 9-582	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.30 __attribute__((always_inline)) function attribute on page 9-549	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.

[g] Only accepted for certain values of `--gnu_version`.

Table 14-1 Supported GNU extensions (continued)

GNU extension	Origin	Modes supported
9.31 __attribute__((const)) function attribute on page 9-550	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.32 __attribute__((constructor[(priority)])) function attribute on page 9-551	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.64 __attribute__((deprecated)) variable attribute on page 9-583	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.34 __attribute__((destructor[(priority)])) function attribute on page 9-553	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.35 __attribute__((format)) function attribute on page 9-554	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.36 __attribute__((format_arg(string-index))) function attribute on page 9-555	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.37 __attribute__((malloc)) function attribute on page 9-556	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.38 __attribute__((noinline)) function attribute on page 9-557	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.39 __attribute__((no_instrument_function)) function attribute on page 9-558	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.41 __attribute__((nonnull)) function attribute on page 9-560	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.42 __attribute__((noreturn)) function attribute on page 9-561	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.44 __attribute__((nothrow)) function attribute on page 9-563	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.58 __attribute__((packed)) type attribute on page 9-577	GCC-specific.	GNU C90, GNU C99, GNU C++.
9.66 __attribute__((packed)) variable attribute on page 9-585	GCC-specific.	C90, C99, GNU C90, GNU C99, GNU C++.
9.46 __attribute__((pure)) function attribute on page 9-565	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.67 __attribute__((section("name"))) variable attribute on page 9-586	GCC-specific.	C99, GNU C90, GNU C99, GNU C++.
9.48 __attribute__((sentinel)) function attribute on page 9-567	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.59 __attribute__((transparent_union)) type attribute on page 9-578	GCC-specific.	GNU C90, GNU C99.
9.68 __attribute__((unused)) variable attribute on page 9-587	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.69 __attribute__((used)) variable attribute on page 9-588	GCC-specific.	C90, C99, GNU C90, GNU C99.
9.70 __attribute__((visibility("visibility_type"))) variable attribute on page 9-589	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.52 __attribute__((warn_unused_result)) on page 9-571	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.

Table 14-1 Supported GNU extensions (continued)

GNU extension	Origin	Modes supported
9.53 __attribute__((weak)) function attribute on page 9-572	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.71 __attribute__((weak)) variable attribute on page 9-590	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.54 __attribute__((weakref("target"))) function attribute on page 9-573	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
9.72 __attribute__((weakref("target"))) variable attribute on page 9-591	GCC-specific.	C90, C99, C++, GNU C90, GNU C99, GNU C++.
Variadic macros	Standard C99.	C90, C99, C++, GNU C90, GNU C99, GNU C++ . [h]
Zero-length arrays	GCC-specific.	GNU C90, GNU C99.

Related references

7.73 --gnu on page 7-350.

Related information

Which GNU language extensions are supported by the ARM Compiler?.

[h] If `--gnu` is specified (GNU modes), GNU-specific syntax applies.

Chapter 15
Standard C Implementation Definition

Provides information required by the ISO C standard for conforming C implementations.

It contains the following sections:

- *15.1 Implementation definition* on page 15-833.
- *15.2 Translation* on page 15-834.
- *15.3 Environment* on page 15-835.
- *15.4 Identifiers* on page 15-837.
- *15.5 Characters* on page 15-838.
- *15.6 Integers* on page 15-840.
- *15.7 Floating-point* on page 15-841.
- *15.8 Arrays and pointers* on page 15-842.
- *15.9 Registers* on page 15-843.
- *15.10 Structures, unions, enumerations, and bitfields* on page 15-844.
- *15.11 Qualifiers* on page 15-848.
- *15.12 Expression evaluation* on page 15-849.
- *15.13 Preprocessing directives* on page 15-850.
- *15.14 Library functions* on page 15-851.
- *15.15 Behaviors considered undefined by the ISO C Standard* on page 15-852.

15.1 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. 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.

Note

The support for the `wctype.h` and `wchar.h` headers excludes wide file operations.

Related references

15.2 Translation on page 15-834.

15.3 Environment on page 15-835.

15.4 Identifiers on page 15-837.

15.5 Characters on page 15-838.

15.6 Integers on page 15-840.

15.7 Floating-point on page 15-841.

15.8 Arrays and pointers on page 15-842.

15.9 Registers on page 15-843.

15.10 Structures, unions, enumerations, and bitfields on page 15-844.

15.11 Qualifiers on page 15-848.

15.12 Expression evaluation on page 15-849.

15.13 Preprocessing directives on page 15-850.

15.14 Library functions on page 15-851.

15.2 Translation

Describes implementation-defined aspects of the ARM C compiler and C library relating to translation, as required by the ISO C standard.

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.

Here:

`error-code`
: Is a number identifying the error type.

`explanation`
: Is a text description of the error.

Related references

Chapter 5 Compiler Diagnostic Messages on page 5-205.

15.3 Environment

Describes implementation-defined aspects of the ARM C compiler and C library relating to environment, as required by the ISO C standard requires.

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:

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.

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.

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()`.

Related references

9.86 #pragma import symbol_name on page 9-606.

15.4 Identifiers

Describes implementation-defined aspects of the ARM C compiler and C library relating to identifiers, as required by the ISO C standard.

The following point applies to the 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.

15.5 Characters

Describes implementation-defined aspects of the ARM C compiler and C library relating to characters, as required by the ISO C standard.

The following points apply to the character sets expected by the compiler:

- 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 change the locale using the `--locale` command-line option if the source file contains non-ASCII characters. If you do not specify `--locale`, the system locale is used.
- The compiler supports multibyte character sets, such as Unicode. You can control this support using the `--[no_]multibyte_chars` options.
- If the source file encoding is UTF-8 or UTF-16, and the file starts with a byte order mark then the compiler ignores the `--[no_]multibyte_chars` and `--locale` options and interprets the file as UTF-8 or UTF-16.
- 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 `NUL` (`\0`) character.
- All integer character constants that contain a single character, or character escape sequence, are represented in both the source and execution character sets.The following table lists the supported character escape codes.

Table 15-1 Character escape codes

Escape sequence	Char value	Description
`\a`	`7`	Attention (bell)
`\b`	`8`	Backspace
`\t`	`9`	Horizontal tab
`\n`	`10`	New line (line feed)
`\v`	`11`	Vertical tab
`\f`	`12`	Form feed
`\r`	`13`	Carriage return

Table 15-1 Character escape codes (continued)

Escape sequence	Char value	Description
`\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 makes the **`char`** signed
  — the `--unsigned_chars` option makes 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.

- Converting multibyte characters into the corresponding wide characters for a wide character constant does not use a locale. This is not relevant to the generic implementation.

15.6 Integers

Describes implementation-defined aspects of the ARM C compiler and C library relating to integers, as required by the ISO C standard.

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.

15.7 Floating-point

Describes implementation-defined aspects of the ARM C compiler and C library relating to floating-point operations, as required by the ISO C standard.

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.

15.8 Arrays and pointers

Describes implementation-defined aspects of the ARM C compiler and C library relating to arrays and pointers, as required by the ISO C standard.

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`.

15.9 Registers

Describes implementation-defined aspects of the ARM C compiler and C library relating to registers, as required by the ISO C standard.

Using the ARM compiler, you can declare any number of local objects to have the storage class **`register`**.

15.10 Structures, unions, enumerations, and bitfields

Describes implementation-defined aspects of the ARM C compiler and C library relating to structures, unions, enumerations, and bitfields, as required by the ISO C standard.

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.

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.

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 was 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.

———————

In strict C, enumerator values must be representable as **int**s. That is, they must be in the range -2147483648 to +2147483647, inclusive. 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 ...
```

Padding and alignment of structures

The following points apply to:

- all C structures
- all C++ structures and classes not using virtual functions or base classes.

Structures can contain padding to ensure that fields are correctly aligned and that the structure itself is correctly aligned. The following diagram 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.

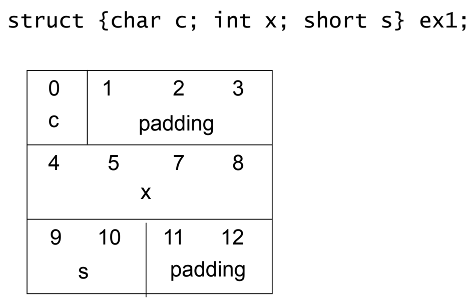

Figure 15-1 Conventional nonpacked structure example

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.

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.

Bitfields

In nonpacked structures, 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. The following figure shows the bitfield allocation for the following example structure:

```
struct X
{
    int x:10;
    char y:8;
};
```

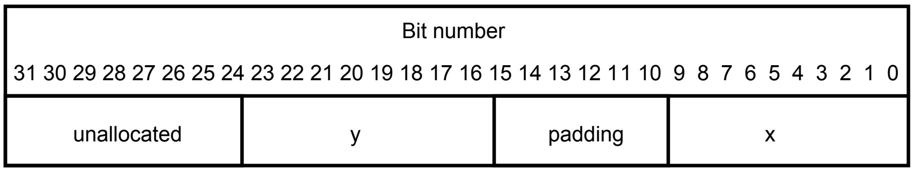

Figure 15-2 Bitfield allocation 1

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, as follows:

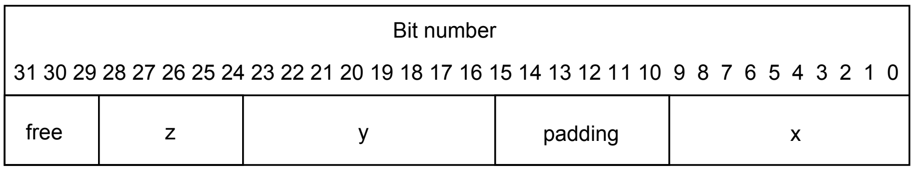

Figure 15-3 Bitfield allocation 2

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.

Note

As an optimization, the compiler might overwrite padding bits in a container with unspecified values when a bitfield is written. This does not affect normal usage of bitfields.

15.11 Qualifiers

Describes implementation-defined aspects of the ARM C compiler and C library relating to qualifiers, as required by the ISO C standard.

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.

15.12 Expression evaluation

Describes implementation-defined aspects of the ARM C compiler and C library relating to expression evaluation, as required by the ISO C standard.

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.

15.13 Preprocessing directives

Describes implementation-defined aspects of the ARM C compiler and C library relating to preprocessing directives, as required by the ISO C standard.

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, the compiler tries to translate the filename to a Windows 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.

Related references

1.2 Source language modes of the compiler on page 1-29.

15.14 Library functions

Describes implementation-defined aspects of the ARM C compiler and C library relating to library functions, as required by the ISO C standard.

The ISO C library variants are listed in *ARM C and C++ Libraries and Floating-Point Support User Guide*.

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>`.

For implementation details of mathematical functions, macros, locale, signals, and input/output see *ARM C and C++ Libraries and Floating-Point Support User Guide*.

Related information

The ARM C and C++ Libraries.

15.15 Behaviors considered undefined by the ISO C Standard

Describes implementation-defined aspects of the ARM C compiler and C library relating to behaviors considered undefined by the ISO C Standard, as required 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.

Chapter 16
Standard C++ Implementation Definition

Lists the C++ language features defined in the ISO/IEC standard for C++, and states whether or not ARM C++ supports that language feature.

The ARM compiler supports the majority of the language features described in the standard.

Note

This documentation does not duplicate information that is part of the standard C implementation

Note

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.

It contains the following sections:

- *16.1 Integral conversion* on page 16-854.
- *16.2 Calling a pure virtual function* on page 16-855.
- *16.3 Major features of language support* on page 16-856.
- *16.4 Standard C++ library implementation definition* on page 16-857.

16.1 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.

16.2 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.

16.3 Major features of language support

The following table shows the major features of the language that this release of ARM C++ supports.

Table 16-1 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 *ARM C and C++ Libraries and Floating-Point Support User Guide*.

16.4 Standard C++ library implementation definition

The Rogue Wave Standard C++ 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 *ARM C and C++ Libraries and Floating-Point Support User Guide*.

The library can be used with user-defined functions to produce target-dependent applications. See *ARM C and C++ Libraries and Floating-Point Support User Guide*.

Chapter 17
C and C++ Compiler Implementation Limits

Describes the implementation limits when using the ARM compiler to compile C and C++.

It contains the following sections:

- *17.1 C++ ISO/IEC standard limits* on page 17-859.
- *17.2 Limits for integral numbers* on page 17-861.
- *17.3 Limits for floating-point numbers* on page 17-862.

17.1 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.

The following table 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 17-1 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.	1 024	memory
Number of initial characters in an external identifier.	1 024	memory
External identifiers in one translation unit.	65 536	memory
Identifiers with block scope declared in one block.	1 024	memory
Macro identifiers simultaneously defined in one translation unit.	65 536	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.	65 536	memory
Characters in a character string literal or wide string literal after concatenation.	65 536	memory
Size of a C or C++ object (including arrays).	262 144	4 294 967 296
Nesting levels of #include file.	256	memory
Case labels for a **`switch`** statement, excluding those for any nested **`switch`** statements.	16 384	memory
Data members in a single class, structure, or union.	16 384	memory
Enumeration constants in a single enumeration.	4 096	memory
Levels of nested class, structure, or union definitions in a single **`struct`** declaration-list.	256	memory
Functions registered by `atexit()`.	32	33
Direct and indirect base classes.	16 384	memory
Direct base classes for a single class.	1 024	memory
Members declared in a single class.	4 096	memory
Final overriding virtual functions in a class, accessible or not.	16 384	memory
Direct and indirect virtual bases of a class.	1 024	memory
Static members of a class.	1 024	memory

Table 17-1 Implementation limits (continued)

Description	Recommended	ARM
Friend declarations in a class.	4 096	memory
Access control declarations in a class.	4 096	memory
Member initializers in a constructor definition.	6 144	memory
Scope qualifications of one identifier.	256	memory
Nested external specifications.	1 024	memory
Template arguments in a template declaration.	1 024	memory
Recursively nested template instantiations.	17	memory
Handlers per try block.	256	memory
Throw specifications on a single function declaration.	256	memory

17.2 Limits for integral numbers

The following table 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 the *ARM Compiler Getting Started Guide*.

Table 17-2 Integer ranges

Constant	Meaning	Value	Hex value
`CHAR_MAX`	Maximum value of **`char`**	255	`0xFF`
`CHAR_MIN`	Minimum value of **`char`**	0	`0x00`
`SCHAR_MAX`	Maximum value of **`signed char`**	127	`0x7F`
`SCHAR_MIN`	Minimum value of **`signed char`**	–128	`0x80`
`UCHAR_MAX`	Maximum value of **`unsigned char`**	255	`0xFF`
`SHRT_MAX`	Maximum value of **`short`**	32 767	`0x7FFF`
`SHRT_MIN`	Minimum value of **`short`**	–32 768	`0x8000`
`USHRT_MAX`	Maximum value of **`unsigned short`**	65 535	`0xFFFF`
`INT_MAX`	Maximum value of **`int`**	2 147 483 647	`0x7FFFFFFF`
`INT_MIN`	Minimum value of **`int`**	–2 147 483 648	`0x80000000`
`LONG_MAX`	Maximum value of **`long`**	2 147 483 647	`0x7FFFFFFF`
`LONG_MIN`	Minimum value of **`long`**	–2 147 483 648	`0x80000000`
`ULONG_MAX`	Maximum value of **`unsigned long`**	4 294 967 295	`0xFFFFFFFF`
`LLONG_MAX`	Maximum value of **`long long`**	9.2E+18	`0x7FFFFFFFFFFFFFFF`
`LLONG_MIN`	Minimum value of **`long long`**	–9.2E+18	`0x8000000000000000`
`ULLONG_MAX`	Maximum value of **`unsigned long long`**	1.8E+19	`0xFFFFFFFFFFFFFFFF`

17.3 Limits for floating-point numbers

This topic describes the characteristics of floating-point numbers.

The following table gives the limits for floating-point numbers. These constants are defined in the `float.h` include file.

Table 17-3 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`**.	1 024
`DBL_MIN_EXP`	Minimum value of base 2 exponent for type **`double`**.	–1 021
`LDBL_MAX_EXP`	Maximum value of base 2 exponent for type **`long double`**.	1 024
`LDBL_MIN_EXP`	Minimum value of base 2 exponent for type **`long double`**.	–1 021
`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

The following table describes other characteristics of floating-point numbers. These constants are also defined in the `float.h` include file.

Table 17-4 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
`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

Table 17-4 Other floating-point characteristics (continued)

Constant	Meaning	Value
`LDBL_MANT_DIG`	Binary digits of precision for type **`long double`**.	53
`FLT_EPSILON`	Smallest positive value of x that 1.0 + x != 1.0 for type **`float`**.	1.19209290e–7F
`DBL_EPSILON`	Smallest positive value of x that 1.0 + x != 1.0 for type **`double`**.	2.2204460492503131e–16
`LDBL_EPSILON`	Smallest positive value of x that 1.0 + x != 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.