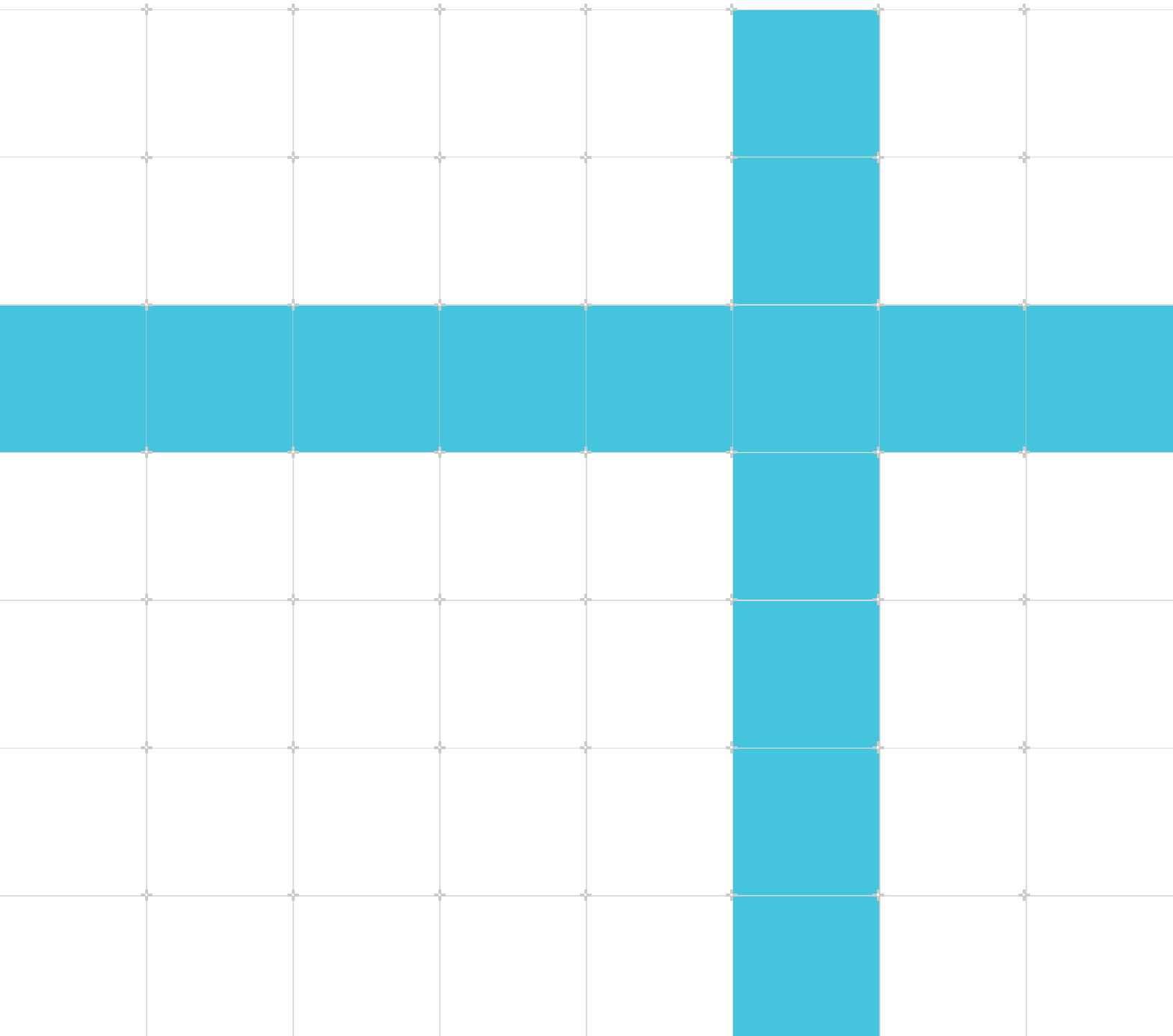
Arm® Compiler for Embedded

Version 6.23

Reference Guide

Non-Confidential
Copyright © 2019–2024 Arm Limited (or its affiliates).
All rights reserved.

Issue 01
101754_6.23_01_en



Arm® Compiler for Embedded Reference Guide

This document is Non-Confidential.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.

This document is protected by copyright and other intellectual property rights.
Arm only permits use of this document if you have reviewed and accepted Arm's Proprietary Notice found at the end of this document.

This document (101754_6.23_01_en) was issued on 2024-10-16. There might be a later issue at https://developer.arm.com/documentation/101754

The product version is 6.23.

See also: Proprietary notice | Product and document information | Useful resources

Start reading

If you prefer, you can skip to the start of the content.

Intended audience

This document is intended for software developers and provides detailed reference information about the features supported in Arm® Compiler for Embedded 6.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com.

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Contents

1. Arm Compiler for Embedded Tools Overview..... 28
1.1 Arm Compiler for Embedded tool command-line syntax..... 28
1.2 Support level definitions..... 30

2. armclang Reference..... 35
2.1 armclang Command-line Options..... 35
2.1.1 -C (armclang)..... 42
2.1.2 -c (armclang)..... 44
2.1.3 -D (armclang)..... 44
2.1.4 -d (armclang)..... 45
2.1.5 -E..... 46
2.1.6 -e..... 48
2.1.7 -faligned-new..... 49
2.1.8 -fbare-metal-pie..... 51
2.1.9 -fbracket-depth=N..... 51
2.1.10 -fcommon, -fno-common..... 52
2.1.11 -fcomplete-member-pointers..... 52
2.1.12 -fdata-sections, -fno-data-sections..... 53
2.1.13 -feliminate-unused-debug-types, -fno-eliminate-unused-debug-types..... 55
2.1.14 -fexceptions, -fno-exceptions..... 56
2.1.15 -ffast-math, -fno-fast-math..... 56
2.1.16 -ffixed-r<N>..... 58
2.1.17 -ffixed-x18..... 60
2.1.18 -ffp-contract..... 61
2.1.19 -ffp-mode..... 62
2.1.20 -ffreestanding..... 66
2.1.21 -ffunction-sections, -fno-function-sections..... 67
2.1.22 -fident, -fno-ident..... 68
2.1.23 -finstrument-functions..... 69
2.1.24 @file..... 70
2.1.25 -fldm-stm, -fno-ldm-stm..... 71
2.1.26 -flto, -fno-lto..... 72

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.27 -fno-builtin....73
2.1.28 -fno-inline-functions....76
2.1.29 -fomit-frame-pointer, -fno-omit-frame-pointer....77
2.1.30 -foptimize-sibling-calls, -fno-optimize-sibling-calls....78
2.1.31 -fpic, -fno-pic....79
2.1.32 -fropi, -fno-ropi....80
2.1.33 -fropi-lowering, -fno-ropi-lowering....81
2.1.34 -frtti, -fno-rtti....82
2.1.35 -frwpi, -fno-rwpi....82
2.1.36 -frwpi-lowering, -fno-rwpi-lowering....83
2.1.37 -fsanitize, -fno-sanitize....84
2.1.38 -fsanitize-ignorelist, -fno-sanitize-ignorelist....87
2.1.39 -fsanitize-minimal-runtime....89
2.1.40 -fsanitize-recover, -fno-sanitize-recover....93
2.1.41 -fsanitize-trap, -fno-sanitize-trap....94
2.1.42 -fshort-enums, -fno-short-enums....96
2.1.43 -fshort-wchar, -fno-short-wchar....97
2.1.44 -fsigned-char, -funsigned-char....98
2.1.45 -fsized-deallocation, -fno-sized-deallocation....99
2.1.46 -fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector....100
2.1.47 -fstack-usage....102
2.1.48 -fstrict-aliasing, -fno-strict-aliasing....104
2.1.49 -fsysv, -fno-sysv....104
2.1.50 -ftls-model....105
2.1.51 -ftrapv....107
2.1.52 -ftrivial-auto-var-init....108
2.1.53 -fvectorize, -fno-vectorize....109
2.1.54 -fvisibility....110
2.1.55 -fwrapv....111
2.1.56 -g, -g<level>, -gdwarf-<version> (armclang)....111
2.1.57 -I....113
2.1.58 -include....114
2.1.59 -isystem <directory>....114
2.1.60 -L....115
2.1.61 -l....116
2.1.62 -M, -MM....116

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.63 -MD, -MMD ... 117
2.1.64 -MF ... 118
2.1.65 -MG ... 119
2.1.66 -MP ... 120
2.1.67 -MT ... 121
2.1.68 -mabi=<name> ... 122
2.1.69 -march ... 123
2.1.70 -marm ... 140
2.1.71 -masm ... 141
2.1.72 -mbig-endian ... 144
2.1.73 -mbranch-protection ... 144
2.1.74 -mcmodel ... 146
2.1.75 -mcmse ... 148
2.1.76 -mcpu ... 150
2.1.77 -mdefault-build-attributes, -mno-default-build-attributes ... 167
2.1.78 -mexecute-only ... 169
2.1.79 -mfloat-abi ... 169
2.1.80 -mfpu ... 170
2.1.81 -mframe-chain ... 173
2.1.82 -mglobal-merge, -mno-global-merge ... 175
2.1.83 -mharden-pac-ret ... 176
2.1.84 -mharden-sls ... 180
2.1.85 -mimplicit-it ... 182
2.1.86 -mlittle-endian ... 183
2.1.87 -mno-neg-immediates ... 184
2.1.88 -moutline, -mno-outline ... 186
2.1.89 -mpixolib ... 189
2.1.90 -mrestrict-it, -mno-restrict-it ... 192
2.1.91 -mthumb ... 193
2.1.92 -mtls-size ... 194
2.1.93 -mtp ... 195
2.1.94 -mtune=target ... 196
2.1.95 -munaligned-access, -mno-unaligned-access ... 197
2.1.96 -nobuiltininc ... 198
2.1.97 -nostdlib ... 200
2.1.98 -nostdlibinc ... 202

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.99 -O (armclang)....203
2.1.100 -o (armclang)....207
2.1.101 -pedantic....207
2.1.102 -pedantic-errors....208
2.1.103 --print-enabled-extensions....209
2.1.104 -resource-dir....212
2.1.105 -Rpass....213
2.1.106 -S (armclang)....216
2.1.107 -save-temps....216
2.1.108 -shared (armclang)....218
2.1.109 -std....218
2.1.110 --target....221
2.1.111 -U....222
2.1.112 -u (armclang)....223
2.1.113 -v (armclang)....223
2.1.114 --verbose (armclang)....224
2.1.115 --version (armclang)....224
2.1.116 --version_number (armclang)....224
2.1.117 --vsn (armclang)....225
2.1.118 -W (armclang)....226
2.1.119 -Wl....228
2.1.120 -Xlinker....228
2.1.121 -x (armclang)....229
2.1.122 -###....230
2.1.123 Examples for the armclang -ffp-contract command-line option....231
2.1.124 Examples for the armclang -fsanitize command-line option....235
2.1.125 Examples for the armclang -mbranch-protection command-line option....240
2.2 Compiler-specific Keywords and Operators....244
2.2.1 __alignof__....245
2.2.2 __asm....246
2.2.3 __declspec attributes....248
2.2.4 __declspec(noinline)....248
2.2.5 __declspec(noreturn)....249
2.2.6 __declspec(nothrow)....250
2.2.7 __inline....251
2.2.8 __promise....251

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.2.9 __unaligned....252
2.2.10 Global named register variables....253
2.3 Compiler-specific Function, Variable, and Type Attributes....257
2.3.1 __attribute__((always_inline)) function attribute....259
2.3.2 __attribute__((cmse_nonsecure_call)) function attribute....260
2.3.3 __attribute__((cmse_nonsecure_entry)) function attribute....261
2.3.4 __attribute__((const)) function attribute....262
2.3.5 __attribute__((constructor(priority))) function attribute....263
2.3.6 __attribute__((format_arg(<string-index>))) function attribute....264
2.3.7 __attribute__((interrupt("type"))) function attribute....265
2.3.8 __attribute__((malloc)) function attribute....267
2.3.9 __attribute__((naked)) function attribute....267
2.3.10 __attribute__((noinline)) function attribute....268
2.3.11 __attribute__((nomerge)) function attribute....269
2.3.12 __attribute__((nonnull)) function attribute....269
2.3.13 __attribute__((no_instrument_function)) function attribute....270
2.3.14 __attribute__((no_sanitize("option"))) function attribute....271
2.3.15 __attribute__((noreturn)) function attribute....272
2.3.16 __attribute__((not_tail_called)) function attribute....272
2.3.17 __attribute__((nothrow)) function attribute....273
2.3.18 __attribute__((optnone)) function attribute....273
2.3.19 __attribute__((pcs("calling_convention"))) function attribute....275
2.3.20 __attribute__((pure)) function attribute....275
2.3.21 __attribute__((section("name"))) function attribute....276
2.3.22 __attribute__((target("options"))) function attribute....277
2.3.23 __attribute__((unused)) function attribute....282
2.3.24 __attribute__((used)) function attribute....283
2.3.25 __attribute__((value_in_regs)) function attribute....284
2.3.26 __attribute__((visibility("visibility_type"))) function attribute....286
2.3.27 __attribute__((weak)) function attribute....287
2.3.28 __attribute__((weakref("target"))) function attribute....287
2.3.29 __attribute__((aligned)) type attribute....288
2.3.30 __attribute__((packed)) type attribute....289
2.3.31 __attribute__((transparent_union)) type attribute....290
2.3.32 __attribute__((alias)) variable attribute....291
2.3.33 __attribute__((aligned)) variable attribute....292

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.3.34 __attribute__((common)), __attribute__((nocommon)) variable attribute....295
2.3.35 __attribute__((deprecated)) variable attribute....296
2.3.36 __attribute__((packed)) variable attribute....296
2.3.37 __attribute__((section("name"))) variable attribute....297
2.3.38 __attribute__((tls_model("model"))) variable attribute....298
2.3.39 __attribute__((uninitialized)) variable attribute....300
2.3.40 __attribute__((unused)) variable attribute....300
2.3.41 __attribute__((used)) variable attribute....301
2.3.42 __attribute__((visibility("visibility_type"))) variable attribute....302
2.3.43 __attribute__((weak)) variable attribute....303
2.3.44 __attribute__((weakref("target"))) variable attribute....303
2.3.45 __attribute__((aligned)) type attribute examples....304
2.4 Compiler-specific Intrinsics....311
2.4.1 __breakpoint intrinsic....311
2.4.2 __current_pc intrinsic....312
2.4.3 __current_sp intrinsic....312
2.4.4 __disable_fiq intrinsic....313
2.4.5 __disable_irq intrinsic....313
2.4.6 __enable_fiq intrinsic....314
2.4.7 __enable_irq intrinsic....315
2.4.8 __force_stores intrinsic....315
2.4.9 __memory_changed intrinsic....315
2.4.10 __schedule_barrier intrinsic....316
2.4.11 __semihost intrinsic....316
2.4.12 __vfp_status intrinsic....318
2.5 Compiler-specific Pragmas....319
2.5.1 #pragma clang diagnostic....319
2.5.2 #pragma clang section....322
2.5.3 #pragma clang system_header....325
2.5.4 #pragma message....325
2.5.5 #pragma once....326
2.5.6 #pragma pack(...)....327
2.5.7 #pragma STDC....329
2.5.8 #pragma unroll[(n)], #pragma unroll_completely....330
2.5.9 #pragma weak symbol, #pragma weak symbol1 = symbol2....331
2.5.10 #pragma STDC FP_CONTRACT examples....331

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.6 Other Compiler-specific Features... 334
2.6.1 ACLE support... 334
2.6.2 Predefined macros... 336
2.6.3 Inline functions... 341
2.6.4 Volatile variables... 342
2.6.5 Half-precision floating-point data types... 342
2.6.6 Half-precision floating-point number format... 346
2.6.7 Half-precision floating-point intrinsics... 347
2.6.8 Library support for _Float16 data type... 348
2.6.9 BFloat16 floating-point number format... 349
2.6.10 TT instruction intrinsics... 350
2.6.11 Non-secure function pointer intrinsics... 353
2.6.12 Supported architecture features... 354
2.6.13 Supported architecture feature combinations for specific processors... 365
2.7 armclang Integrated Assembler... 368
2.7.1 Syntax of assembly files for integrated assembler... 369
2.7.2 Assembly expressions... 370
2.7.3 Alignment directives... 375
2.7.4 Data definition directives... 377
2.7.5 String definition directives... 380
2.7.6 Floating-point data definition directives... 382
2.7.7 Section directives... 382
2.7.8 Conditional assembly directives... 387
2.7.9 Macro directives... 389
2.7.10 Symbol binding directives... 390
2.7.11 Org directive... 391
2.7.12 AArch32 target selection directives... 392
2.7.13 AArch64 target selection directives... 395
2.7.14 Space-filling directives... 397
2.7.15 Type directive... 398
2.7.16 Call Frame Information directives... 399
2.7.17 A32 and T32 instruction substitutions... 399
2.7.18 Integrated assembler support for the CSDB instruction... 400
2.8 armclang inline assembler... 401
2.8.1 Inline Assembly... 401
2.8.2 File-scope inline assembly... 404

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.8.3 Inline assembly statements within a function....406
2.8.4 Inline assembly constraint strings....412
2.8.5 Inline assembly template modifiers....418
2.8.6 Forcing inline assembly operands into specific registers....421
2.8.7 Symbol references and branches into and out of inline assembly....421
2.8.8 Duplication of labels in inline assembly statements....423

3. armlink Reference....425
3.1 armlink Command-line Options....425
3.1.1 --any_contingency....425
3.1.2 --any_placement=algorithm....426
3.1.3 --any_sort_order=order....427
3.1.4 --api, --no_api....428
3.1.5 --autoat, --no_autoat....429
3.1.6 --bare_metal_pie....430
3.1.7 --bare_metal_sysv....431
3.1.8 --be8....431
3.1.9 --be32....431
3.1.10 --bestdebug, --no_bestdebug....432
3.1.11 --blx_arm_thumb, --no_blx_arm_thumb....433
3.1.12 --blx_thumb_arm, --no_blx_thumb_arm....433
3.1.13 --branchnop, --no_branchnop....434
3.1.14 --callgraph, --no_callgraph....434
3.1.15 --callgraph_file=filename....436
3.1.16 --callgraph_output=fmt....437
3.1.17 --callgraph_subset=symbol[,symbol,...]....437
3.1.18 --cgfile=type....438
3.1.19 --cgsymbol=type....439
3.1.20 --cgundefined=type....440
3.1.21 --check_pac_mismatch....440
3.1.22 --comment_section, --no_comment_section....441
3.1.23 --cppinit, --no_cppinit....442
3.1.24 --cpu=list (armlink)....442
3.1.25 --cpu=name (armlink)....443
3.1.26 --crosser_veneershare, --no_crosser_veneershare....446
3.1.27 --dangling-debug-address=address....446

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.28 Example showing the effect of using --dangling-debug-address....447
3.1.29 --datacompressor=opt....449
3.1.30 --debug, --no_debug....450
3.1.31 --diag_error=tag[,tag,...] (armlink)....450
3.1.32 --diag_remark=tag[,tag,...] (armlink)....451
3.1.33 --diag_style=arm|ide|gnu (armlink)....451
3.1.34 --diag_suppress=tag[,tag,...] (armlink)....452
3.1.35 --diag_warning=tag[,tag,...] (armlink)....453
3.1.36 --dynamic_linker=name....454
3.1.37 --eager_load_debug, --no_eager_load_debug....454
3.1.38 --edit=file_list....455
3.1.39 --eh_frame_hdr....456
3.1.40 --elf-output-format....456
3.1.41 --emit_debug_overlay_relocs....458
3.1.42 --emit_debug_overlay_section....459
3.1.43 --emit_non_debug_relocs....459
3.1.44 --emit_relocs....460
3.1.45 --entry=location....460
3.1.46 --errors=filename....463
3.1.47 --exceptions, --no_exceptions....464
3.1.48 --export_all, --no_export_all....464
3.1.49 --export_dynamic, --no_export_dynamic....465
3.1.50 --filtercomment, --no_filtercomment....466
3.1.51 --fini=symbol....466
3.1.52 --first=section_id....467
3.1.53 --force_explicit_attr....468
3.1.54 --force_so_throw, --no_force_so_throw....468
3.1.55 --fpic....469
3.1.56 --fpu=list (armlink)....469
3.1.57 --fpu=name (armlink)....470
3.1.58 --got=type....471
3.1.59 --gnu_linker_defined_syms....471
3.1.60 --help (armlink)....472
3.1.61 --import_cmse_lib_in=filename....473
3.1.62 --import_cmse_lib_out=filename....473
3.1.63 --import_unresolved, --no_import_unresolved....474

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.64 --info=topic[,topic,...] (armlink)..... 474
3.1.65 --info_lib_prefix=opt..... 478
3.1.66 --init=symbol..... 479
3.1.67 --inline, --no_inline..... 479
3.1.68 --inline_type=type..... 480
3.1.69 --inlineveneer, --no_inlineveneer..... 481
3.1.70 input-file-list (armlink)..... 481
3.1.71 --keep=section_id (armlink)..... 482
3.1.72 --keep_intermediate..... 487
3.1.73 --largeregions, --no_largeregions..... 488
3.1.74 --last=section_id..... 489
3.1.75 --legacyalign, --no_legacyalign..... 490
3.1.76 --libpath=pathlist..... 491
3.1.77 --library=name..... 491
3.1.78 --library_security=protection..... 492
3.1.79 --library_type=lib..... 495
3.1.80 --list=filename..... 495
3.1.81 --list_mapping_symbols, --no_list_mapping_symbols..... 496
3.1.82 --load_addr_map_info, --no_load_addr_map_info..... 497
3.1.83 --locals, --no_locals..... 497
3.1.84 --lto, --no_lto..... 498
3.1.85 --lto_keep_all_symbols, --no_lto_keep_all_symbols..... 500
3.1.86 --lto_intermediate_filename..... 501
3.1.87 --lto_level..... 501
3.1.88 --lto_relocation_model..... 503
3.1.89 --mangled, --unmangled..... 504
3.1.90 --map, --no_map..... 505
3.1.91 --max_er_extension=size..... 505
3.1.92 --max_veneer_passes=value..... 506
3.1.93 --max_visibility=type..... 506
3.1.94 --merge, --no_merge..... 507
3.1.95 --merge_litpools, --no_merge_litpools..... 508
3.1.96 --muldefweak, --no_muldefweak..... 508
3.1.97 -o filename, --output=filename (armlink)..... 509
3.1.98 --output_float_abi=option..... 509
3.1.99 --overlay_veneers..... 511

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.100 --override_visibility... 512
3.1.101 -Omax (armlink)... 512
3.1.102 -Omin (armlink)... 513
3.1.103 --pad=num... 513
3.1.104 --paged... 514
3.1.105 --pagesize=pagesize... 514
3.1.106 --partial... 515
3.1.107 --pie... 515
3.1.108 --piveneer, --no_piveneer... 516
3.1.109 --pixolib... 516
3.1.110 --predefine="string"... 518
3.1.111 --preinit, --no_preinit... 520
3.1.112 --privacy (armlink)... 520
3.1.113 --ref_cpp_init, --no_ref_cpp_init... 521
3.1.114 --ref_pre_init, --no_ref_pre_init... 522
3.1.115 --reloc... 522
3.1.116 --remarks... 523
3.1.117 --remove, --no_remove... 524
3.1.118 --require-bti... 524
3.1.119 --ro_base=address... 525
3.1.120 --ropi... 526
3.1.121 --rosplit... 527
3.1.122 --rw_base=address... 527
3.1.123 --rwpi... 528
3.1.124 --scanlib, --no_scanlib... 529
3.1.125 --scatter=filename... 529
3.1.126 --scatterload-enabled, --no-scatterload-enabled... 531
3.1.127 --section_index_display=type... 531
3.1.128 --shared... 532
3.1.129 --show_cmdline (armlink)... 533
3.1.130 --show_full_path... 534
3.1.131 --show_parent_lib... 534
3.1.132 --show_sec_idx... 535
3.1.133 --soname=name... 535
3.1.134 --sort=algorithm... 536
3.1.135 --split... 538

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.136 --startup=symbol, --no_startup ... 539
3.1.137 --stdlib ... 539
3.1.138 --strict ... 540
3.1.139 --strict_flags, --no_strict_flags ... 541
3.1.140 --strict_ph, --no_strict_ph ... 541
3.1.141 --strict_preserve8_require8 ... 542
3.1.142 --strict_relocations, --no_strict_relocations ... 543
3.1.143 --strict_symbols, --no_strict_symbols ... 543
3.1.144 --strict_visibility, --no_strict_visibility ... 544
3.1.145 --summary_stderr, --no_summary_stderr ... 545
3.1.146 --symbols, --no_symbols ... 546
3.1.147 --symdefs=filename ... 546
3.1.148 --symver_script=filename ... 547
3.1.149 --symver_soname ... 547
3.1.150 --sysv ... 548
3.1.151 --tailreorder, --no_tailreorder ... 549
3.1.152 --tiebreaker=option ... 550
3.1.153 --unaligned_access, --no_unaligned_access (armlink) ... 551
3.1.154 --undefined=symbol ... 551
3.1.155 --undefined_and_export=symbol ... 552
3.1.156 --unresolved=symbol ... 553
3.1.157 --use_definition_visibility ... 553
3.1.158 --userlibpath=pathlist ... 554
3.1.159 --veneerinject, --no_veneerinject ... 554
3.1.160 --veneer_inject_type=type ... 555
3.1.161 --veneer_pool_size=size ... 556
3.1.162 --veneershare, --no_veneershare ... 557
3.1.163 --verbose ... 557
3.1.164 --version_number (armlink) ... 558
3.1.165 --via=filename (armlink) ... 558
3.1.166 --vsn (armlink) ... 559
3.1.167 --xo_base=address ... 560
3.1.168 --xref, --no_xref ... 561
3.1.169 --xrefdbg, --no_xrefdbg ... 561
3.1.170 --xreffrom, --xrefto ... 562
3.1.171 --zi_base=address ... 563

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.172 Examples for the armlink --elf-output-format=gnu command-line option... 563
3.2 Linking Models Supported by armlink... 567
3.2.1 Overview of linking models... 567
3.2.2 Bare-metal linking model overview... 567
3.2.3 Partial linking model overview... 569
3.2.4 SysV linking model overview... 569
3.3 Image Structure and Generation... 570
3.3.1 The structure of an Arm ELF image... 570
3.3.2 Simple images... 582
3.3.3 Section placement with the linker... 590
3.3.4 Linker support for creating demand-paged files... 595
3.3.5 Linker reordering of execution regions containing T32 code... 597
3.3.6 Linker-generated veneers... 597
3.3.7 Command-line options used to control the generation of C++ exception tables... 602
3.3.8 Weak references and definitions... 603
3.3.9 How the linker performs library searching, selection, and scanning... 606
3.3.10 How the linker searches for the Arm standard libraries... 607
3.3.11 Specifying user libraries when linking... 608
3.3.12 How the linker resolves references... 609
3.3.13 The strict family of linker options... 610
3.4 Linker Optimization Features... 610
3.4.1 Elimination of common section groups... 610
3.4.2 Elimination of unused sections... 611
3.4.3 Optimization with RW data compression... 612
3.4.4 Function inlining with the linker... 615
3.4.5 Factors that influence function inlining... 616
3.4.6 About branches that optimize to a NOP... 618
3.4.7 Linker reordering of tail calling sections... 618
3.4.8 Restrictions on reordering of tail calling sections... 619
3.4.9 Linker merging of comment sections... 619
3.4.10 Merging identical constants... 620
3.5 Accessing and Managing Symbols with armlink... 622
3.5.1 About mapping symbols... 622
3.5.2 Linker-defined symbols... 623
3.5.3 Region-related symbols... 624
3.5.4 Section-related symbols... 633

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.5.5 Access symbols in another image... 635
3.5.6 Edit the symbol tables with a steering file... 639
3.5.7 Use of $Super$$ and $Sub$$ to patch symbol definitions... 642
3.6 Scatter-loading Features... 643
3.6.1 The scatter-loading mechanism... 643
3.6.2 Root region and the initial entry point... 651
3.6.3 Example of how to explicitly place a named section with scatter-loading... 669
3.6.4 Manual placement of unassigned sections... 671
3.6.5 Placing veneers with a scatter file... 685
3.6.6 Placement of CMSE veneer sections for a Secure image... 686
3.6.7 Reserving an empty block of memory... 688
3.6.8 Placement of Arm C and C++ library code... 690
3.6.9 Alignment of regions to page boundaries... 693
3.6.10 Alignment of execution regions and input sections... 695
3.6.11 Preprocessing a scatter file... 696
3.6.12 Example of using expression evaluation in a scatter file to avoid padding... 697
3.6.13 Equivalent scatter-loading descriptions for simple images... 698
3.6.14 How the linker resolves multiple matches when processing scatter files... 706
3.6.15 How the linker resolves path names when processing scatter files... 708
3.6.16 Scatter file to ELF mapping... 709
3.7 Scatter File Syntax... 711
3.7.1 BNF notation used in scatter-loading description syntax... 711
3.7.2 Syntax of a scatter file... 712
3.7.3 Load region descriptions... 713
3.7.4 Execution region descriptions... 720
3.7.5 Input section descriptions... 729
3.7.6 Expression evaluation in scatter files... 735
3.8 SysV Shared Libraries and Executables... 746
3.8.1 SysV linking model... 746
3.8.2 Symbol versioning... 756
3.9 Linker Steering File Command Reference... 759
3.9.1 EXPORT steering file command... 760
3.9.2 HIDE steering file command... 761
3.9.3 IMPORT steering file command... 762
3.9.4 RENAME steering file command... 763
3.9.5 REQUIRE steering file command... 764

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.9.6 RESOLVE steering file command....765
3.9.7 SHOW steering file command....766

4. fromelf Reference....767
4.1 fromelf Command-line Options....767
4.1.1 --base [[object_file::]load_region_ID=]num....767
4.1.2 --bin....768
4.1.3 --bincombined....769
4.1.4 --bincombined_base=address....771
4.1.5 --bincombined_padding=size,num....771
4.1.6 --cad....772
4.1.7 --cadcombined....774
4.1.8 --compare=option[,option,...]....775
4.1.9 --continue_on_error....776
4.1.10 --coprocN=value (fromelf)....776
4.1.11 --coprocnvalue examples....777
4.1.12 --cpu=list (fromelf)....779
4.1.13 --cpu=name (fromelf)....779
4.1.14 --datasymbols....784
4.1.15 --debugonly....784
4.1.16 --decode_build_attributes....785
4.1.17 --diag_error=tag[,tag,...] (fromelf)....786
4.1.18 --diag_remark=tag[,tag,...] (fromelf)....787
4.1.19 --diag_style=arm|ide|gnu (fromelf)....787
4.1.20 --diag_suppress=tag[,tag,...] (fromelf)....788
4.1.21 --diag_warning=tag[,tag,...] (fromelf)....788
4.1.22 --disassemble....789
4.1.23 --dump_build_attributes....790
4.1.24 --elf....791
4.1.25 --emit=option[,option,...]....791
4.1.26 --expandarrays....794
4.1.27 --extract_build_attributes....795
4.1.28 --fieldoffsets....796
4.1.29 --fpu=list (fromelf)....798
4.1.30 --fpu=name (fromelf)....799
4.1.31 --globalize=option[,option,...]....800

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.32 --help (fromelf)....800
4.1.33 --hide=option[,option,...]....801
4.1.34 --hide_and_localize=option[,option,...]....802
4.1.35 --i32....802
4.1.36 --i32combined....803
4.1.37 --ignore_section=option[,option,...]....804
4.1.38 --ignore_symbol=option[,option,...]....805
4.1.39 --in_place....806
4.1.40 --info=topic[,topic,...] (fromelf)....806
4.1.41 input_file (fromelf)....808
4.1.42 --interleave=option....809
4.1.43 --linkview, --no_linkview....810
4.1.44 --localize=option[,option,...]....811
4.1.45 --m32....812
4.1.46 --m32combined....813
4.1.47 --only=section_name....814
4.1.48 --output=destination....815
4.1.49 --privacy (fromelf)....816
4.1.50 --qualify....816
4.1.51 --relax_section=option[,option,...]....817
4.1.52 --relax_symbol=option[,option,...]....818
4.1.53 --rename=option[,option,...]....819
4.1.54 --select=select_options....820
4.1.55 --show=option[,option,...]....821
4.1.56 --show_and_globalize=option[,option,...]....822
4.1.57 --show_cmdline (fromelf)....822
4.1.58 --source_directory=path....823
4.1.59 --strip=option[,option,...]....823
4.1.60 --symbolversions, --no_symbolversions....825
4.1.61 --text....826
4.1.62 --version_number (fromelf)....829
4.1.63 --vhx....830
4.1.64 --via=file (fromelf)....831
4.1.65 --vsn (fromelf)....831
4.1.66 -w....833
4.1.67 --wide64bit....834

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.68 --widthxbanks....834

5. armar Reference....837
5.1 armar Command-line Options....837
5.1.1 archive....837
5.1.2 -a pos_name....837
5.1.3 -b pos_name....838
5.1.4 -c (armar)....838
5.1.5 -C (armar)....839
5.1.6 --create....839
5.1.7 -d (armar)....840
5.1.8 --debug_symbols....840
5.1.9 --diag_error=tag[,tag,...] (armar)....841
5.1.10 --diag_remark=tag[,tag,...] (armar)....841
5.1.11 --diag_style=arm|ide|gnu (armar)....842
5.1.12 --diag_suppress=tag[,tag,...] (armar)....842
5.1.13 --diag_warning=tag[,tag,...] (armar)....843
5.1.14 --entries....844
5.1.15 file_list....845
5.1.16 --help (armar)....845
5.1.17 -i pos_name....846
5.1.18 -m pos_name (armar)....846
5.1.19 -n....847
5.1.20 --new_files_only....847
5.1.21 -p....848
5.1.22 -r....848
5.1.23 -s (armar)....849
5.1.24 --show_cmdline (armar)....850
5.1.25 --sizes....851
5.1.26 -t....851
5.1.27 -T....852
5.1.28 -u (armar)....852
5.1.29 -v (armar)....853
5.1.30 --version_number (armar)....854
5.1.31 --via=filename (armar)....854
5.1.32 --vsn (armar)....855

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.1.33 -x (armar)....856
5.1.34 --zs....857
5.1.35 --zt....857

6. armasm Legacy Assembler Reference....859
6.1 armasm Command-line Options....859
6.1.1 --16....859
6.1.2 --32....860
6.1.3 --apcs=qualifier...qualifier....860
6.1.4 --arm....862
6.1.5 --arm_only....863
6.1.6 --bi....863
6.1.7 --bigend....863
6.1.8 --brief_diagnostics, --no_brief_diagnostics....864
6.1.9 --checkreglist....864
6.1.10 --cpreproc....865
6.1.11 --cpreproc_opts=option[,option,...]....865
6.1.12 --cpu=list (armasm)....867
6.1.13 --cpu=name (armasm)....867
6.1.14 --debug....870
6.1.15 --depend=dependfile....871
6.1.16 --depend_format=string....871
6.1.17 --diag_error=tag[,tag,...] (armasm)....872
6.1.18 --diag_remark=tag[,tag,...] (armasm)....873
6.1.19 --diag_style=arm|ide|gnu (armasm)....873
6.1.20 --diag_suppress=tag[,tag,...] (armasm)....874
6.1.21 --diag_warning=tag[,tag,...] (armasm)....875
6.1.22 --dllexport_all....876
6.1.23 --dwarf2....876
6.1.24 --dwarf3....877
6.1.25 --errors=errorfile....877
6.1.26 --exceptions, --no_exceptions (armasm)....877
6.1.27 --exceptions_unwind, --no_exceptions_unwind....878
6.1.28 --execstack, --no_execstack....879
6.1.29 --execute_only....879
6.1.30 --fpmode=model....880

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.1.31 --fpu=list (armasm)....881
6.1.32 --fpu=name (armasm)....882
6.1.33 -g (armasm)....882
6.1.34 --help (armasm)....883
6.1.35 -idir[,dir, ...]....883
6.1.36 --keep (armasm)....884
6.1.37 --length=n....884
6.1.38 --li....884
6.1.39 --library_type=lib (armasm)....885
6.1.40 --list=file....885
6.1.41 --list=....886
6.1.42 --littleend....887
6.1.43 -m (armasm)....887
6.1.44 --maxcache=n....887
6.1.45 --md....888
6.1.46 --no_code_gen....888
6.1.47 --no_esc....888
6.1.48 --no_hide_all....889
6.1.49 --no_regs....889
6.1.50 --no_terse....890
6.1.51 --no_warn....890
6.1.52 -o filename (armasm)....890
6.1.53 --pd....891
6.1.54 --predefine "directive"....891
6.1.55 --regnames....892
6.1.56 --report-if-not-wysiwyg....893
6.1.57 --show_cmdline (armasm)....893
6.1.58 --thumb....894
6.1.59 --unaligned_access, --no_unaligned_access (armasm)....894
6.1.60 --unsafe....895
6.1.61 --untyped_local_labels....895
6.1.62 --version_number (armasm)....896
6.1.63 --via=filename (armasm)....896
6.1.64 --vsn (armasm)....897
6.1.65 --width=n....898
6.1.66 --xref....898

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.2 Structure of armasm Assembly Language Modules....899
6.2.1 Syntax of source lines in armasm syntax assembly language....899
6.2.2 Literals....901
6.2.3 ELF sections and the AREA directive....901
6.2.4 An example armasm syntax assembly language module....902
6.3 Writing A32/T32 Instructions in armasm Syntax Assembly Language....904
6.3.1 About the Unified Assembler Language....905
6.3.2 Syntax differences between UAL and A64 assembly language....905
6.3.3 Register usage in subroutine calls....906
6.3.4 Load immediate values....907
6.3.5 Load immediate values using MOV and MVN....908
6.3.6 Load immediate values using MOV32....910
6.3.7 Load immediate values using LDR Rd, =const....911
6.3.8 Literal pools....912
6.3.9 Load addresses into registers....913
6.3.10 Load addresses to a register using ADR....913
6.3.11 Load addresses to a register using ADRL....915
6.3.12 Load addresses to a register using LDR Rd, =label....916
6.3.13 Other ways to load and store registers....918
6.3.14 Load and store multiple register instructions....918
6.3.15 Load and store multiple register instructions in A32 and T32....919
6.3.16 Stack implementation using LDM and STM....920
6.3.17 Stack operations for nested subroutines....921
6.3.18 Block copy with LDM and STM....922
6.3.19 Memory accesses....924
6.3.20 The Read-Modify-Write operation....924
6.3.21 Optional hash with immediate constants....925
6.3.22 Use of macros....925
6.3.23 Test-and-branch macro example....926
6.3.24 Unsigned integer division macro example....926
6.3.25 Instruction and directive relocations....928
6.3.26 Symbol versions....930
6.3.27 Frame directives....930
6.3.28 Exception tables and Unwind tables....931
6.3.29 Extended notation extension for floating-point code....932
6.3.30 Extended notation extension for Advanced SIMD....932

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.3.31 Normal Advanced SIMD instructions....933
6.4 Using armasm....933
6.4.1 Specify command-line options with an environment variable....934
6.4.2 Using stdin to input source code to the assembler....935
6.4.3 Built-in variables and constants....936
6.4.4 Identifying versions of armasm in source code....939
6.4.5 Diagnostic messages....940
6.4.6 Interlocks diagnostics....940
6.4.7 Automatic IT block generation in T32 code....941
6.4.8 T32 branch target alignment....941
6.4.9 T32 code size diagnostics....942
6.4.10 A32 and T32 instruction portability diagnostics....942
6.4.11 T32 instruction width diagnostics....942
6.4.12 Two pass assembler diagnostics....943
6.4.13 Using the C preprocessor....943
6.4.14 Address alignment in A32/T32 code....945
6.4.15 Address alignment in A64 code....946
6.4.16 Instruction width selection in T32 code....946
6.5 Symbols, Literals, Expressions, and Operators in armasm Assembly Language....947
6.5.1 Symbol naming rules....947
6.5.2 Variables....948
6.5.3 Numeric constants....949
6.5.4 Assembly time substitution of variables....949
6.5.5 Register-relative and PC-relative expressions....950
6.5.6 Labels....951
6.5.7 Labels for PC-relative addresses....951
6.5.8 Labels for register-relative addresses....952
6.5.9 Labels for absolute addresses....953
6.5.10 Numeric local labels....953
6.5.11 Syntax of numeric local labels....954
6.5.12 String expressions....955
6.5.13 String literals....956
6.5.14 Numeric expressions....956
6.5.15 Syntax of numeric literals....957
6.5.16 Syntax of floating-point literals....958
6.5.17 Logical expressions....959

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.5.18 Logical literals....960
6.5.19 Unary operators....960
6.5.20 Binary operators....961
6.5.21 Multiplicative operators....961
6.5.22 String manipulation operators....962
6.5.23 Shift operators....962
6.5.24 Addition, subtraction, and logical operators....963
6.5.25 Relational operators....963
6.5.26 Boolean operators....964
6.5.27 Operator precedence....965
6.5.28 Difference between operator precedence in assembly language and C....965
6.6 armasm Directives Reference....966
6.6.1 Alphabetical list of directives armasm assembly language directives....967
6.6.2 About armasm assembly language control directives....967
6.6.3 About frame directives....968
6.6.4 Directives that can be omitted in pass 2 of the assembler....969
6.6.5 ALIAS directive....971
6.6.6 ALIGN directive....972
6.6.7 AREA directive....974
6.6.8 ARM or CODE32 directive....978
6.6.9 ASSERT directive....979
6.6.10 ATTR directive....979
6.6.11 CN directive....981
6.6.12 CODE16 directive....981
6.6.13 COMMON directive....982
6.6.14 CP directive....983
6.6.15 DATA directive....984
6.6.16 DCB directive....984
6.6.17 DCD and DCDU directives....985
6.6.18 DCDO directive....986
6.6.19 DCFD and DCFDU directives....986
6.6.20 DCFS and DCFSU directives....987
6.6.21 DCI directive....988
6.6.22 DCQ and DCQU directives....989
6.6.23 DCW and DCWU directives....990
6.6.24 END directive....991

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.25 ENDFUNC or ENDP directive....991
6.6.26 ENTRY directive....991
6.6.27 EQU directive....992
6.6.28 EXPORT or GLOBAL directive....993
6.6.29 EXPORTAS directive....995
6.6.30 FIELD directive....996
6.6.31 FRAME ADDRESS directive....997
6.6.32 FRAME POP directive....998
6.6.33 FRAME PUSH directive....999
6.6.34 FRAME REGISTER directive....1000
6.6.35 FRAME RESTORE directive....1001
6.6.36 FRAME RETURN ADDRESS directive....1001
6.6.37 FRAME SAVE directive....1002
6.6.38 FRAME STATE REMEMBER directive....1003
6.6.39 FRAME STATE RESTORE directive....1003
6.6.40 FRAME UNWIND ON directive....1004
6.6.41 FRAME UNWIND OFF directive....1004
6.6.42 FUNCTION or PROC directive....1005
6.6.43 GBLA, GBLL, and GBLS directives....1006
6.6.44 GET or INCLUDE directive....1007
6.6.45 IF, ELSE, ENDIF, and ELIF directives....1008
6.6.46 IMPORT and EXTERN directives....1010
6.6.47 INCBIN directive....1012
6.6.48 INFO directive....1013
6.6.49 KEEP directive....1014
6.6.50 LCLA, LCLL, and LCLS directives....1014
6.6.51 LTORG directive....1015
6.6.52 MACRO and MEND directives....1016
6.6.53 MAP directive....1019
6.6.54 MEXIT directive....1020
6.6.55 NOFP directive....1020
6.6.56 OPT directive....1021
6.6.57 QN, DN, and SN directives....1022
6.6.58 RELOC directive....1023
6.6.59 REQUIRE directive....1025
6.6.60 REQUIRE8 and PRESERVE8 directives....1025

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.61 RLIST directive....1026
6.6.62 RN directive....1027
6.6.63 ROUT directive....1027
6.6.64 SETA, SETL, and SETS directives....1028
6.6.65 SPACE and FILL directives....1030
6.6.66 THUMB directive....1031
6.6.67 TTL and SUBT directives....1031
6.6.68 WHILE and WEND directives....1032
6.6.69 WN and XN....1033
6.7 armasm-Specific A32 and T32 Instruction Set Features....1034
6.7.1 armasm support for the CSDB instruction....1034
6.7.2 ADRL pseudo-instruction....1035
6.7.3 CPY pseudo-instruction....1036
6.7.4 IT instruction (A32)....1037
6.7.5 LDR pseudo-instruction....1038
6.7.6 MOV32 pseudo-instruction....1040
6.7.7 NEG pseudo-instruction....1041
6.7.8 SEVL instruction (A32)....1042
6.7.9 UND pseudo-instruction....1042

A. Supporting Information Appendix....1044
A.1 Standard C Implementation Definition....1044
A.1.1 Implementation definition (ISO C Standard)....1044
A.1.2 Translation....1045
A.1.3 Translation limits....1045
A.1.4 Environment....1046
A.1.5 Identifiers....1048
A.1.6 Characters....1048
A.1.7 Integers....1050
A.1.8 Floating-point....1051
A.1.9 Arrays and pointers....1053
A.1.10 Hints....1053
A.1.11 Structures, unions, enumerations, and bitfields....1053
A.1.12 Qualifiers....1054
A.1.13 Preprocessing directives (ISO C Standard)....1054
A.1.14 Library functions....1056

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A.1.15 Architecture.... 1061
A.2 Standard C++ Implementation Definition.... 1065
A.2.1 Implementation definition (ISO C++ Standard).... 1066
A.2.2 General.... 1066
A.2.3 Lexical conventions.... 1067
A.2.4 Basic concepts.... 1068
A.2.5 Standard conversions.... 1069
A.2.6 Expressions.... 1069
A.2.7 Declarations.... 1071
A.2.8 Declarators.... 1071
A.2.9 Templates.... 1072
A.2.10 Exception handling.... 1072
A.2.11 Preprocessing directives (ISO C++ Standard).... 1072
A.2.12 Library introduction.... 1073
A.2.13 Language support library.... 1074
A.2.14 General utilities library.... 1074
A.2.15 Strings library.... 1075
A.2.16 Numerics library.... 1075
A.2.17 Localization library.... 1076
A.2.18 Containers library.... 1077
A.2.19 Input/output library.... 1077
A.2.20 Regular expressions library.... 1078
A.2.21 Atomic operations library.... 1079
A.2.22 Thread support library.... 1079
A.2.23 Implementation quantities.... 1079
A.3 Via File Syntax.... 1082
A.3.1 Via file syntax rules.... 1083

Proprietary notice.... 1085

Product and document information.... 1087
Product status.... 1087
Revision history.... 1087
Conventions.... 1102

Useful resources.... 1104

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

1. Arm Compiler for Embedded Tools Overview

Arm® Compiler for Embedded comprises tools to create ELF object files, ELF image files, and library files. You can also modify ELF object and image files, and display information on those files.

Also, support level indicators identify the current level of support for various features described in this document. See Support level definitions for more information.


Note

Comments inside source files and header files that are provided by Arm might not be accurate and must not be treated as documentation about the product.

1.1 Arm Compiler for Embedded tool command-line syntax

The Arm® Compiler for Embedded tool commands can accept many input files together with options that determine how to process the files.

Syntax

For the `armclang`, `armasm`, `armlink`, or `fromelf` tools:

```
<tool_name> <options> <input-file-list>
```

For the `armar` tool:

```
armar <options> <archive> [<file_list>]
```

Parameters

`<tool_name>`

Is one of `armclang`, `armasm`, `armlink`, or `fromelf`.

`<options>`

The tool command-line options.

`<input-file-list>`

The input files depend on the tool:

`armclang`

A space-separated list of C, C++, or GNU syntax assembler files.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

armasm

A space-separated list of assembler files containing legacy Arm assembler.


Note

The armasm legacy assembler is deprecated, and it has not been updated since Arm Compiler 6.10. Also, armasm does not support:

- Armv8.4-A or later architectures.
- Certain backported options in Armv8.2-A and Armv8.3-A.
- Assembling SVE instructions.
- Armv8.1-M or later architectures, including MVE.
- All versions of the Armv8-R architecture.

As a reminder, armasm always reports the deprecation warning A1950W. To suppress this message, specify the --diag_suppress=1950 option.

armlink

A space-separated list of objects, libraries, or symbol definitions (symdefs) files.


Note

Some armlink options, such as --keep, require parentheses within their values. On Unix systems your shell typically requires the parentheses to be escaped with backslashes. Alternatively, enclose the complete section specifier in double quotes, for example:

```
--keep="foo.o(Premier*)"
```

fromelf

The ELF file or library file to be processed. When some options are used, multiple input files can be specified.

<archive>

The filename of the library. A library file must always be specified with armar.

<file_list>

The list of files to be processed by armar.

Related information

armclang Command-line Options on page 35
armasm Command-line Options on page 859
input-file-list (armlink) on page 481
armlink Command-line Options on page 425
fromelf Command-line Options on page 767
input_file (fromelf) on page 808
armar Command-line Options on page 837

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

archive on page 837
file_list on page 845

1.2 Support level definitions

Arm® Compiler for Embedded 6 is built on Clang and LLVM technology. Therefore, it has more functionality than the set of product features described in the documentation.

Arm welcomes feedback regarding the use of all Arm Compiler for Embedded 6 features, and intends to support users to a level that is appropriate for that feature. You can contact support at https://developer.arm.com/support.

The following definitions clarify the levels of support and guarantees on functionality that are expected from these features.

Identification in the documentation

All features that are documented in the Arm Compiler for Embedded 6 documentation are product features, except where explicitly stated. The limitations of non-product features are explicitly stated.

Product features

Product features are suitable for use in a production environment. The functionality is well-tested, and is expected to be stable across feature and update releases.

- Arm intends to give advance notice of significant functionality changes to product features.
- If you have a support and maintenance contract, Arm provides full support for use of all product features.
- Arm welcomes feedback on product features.
- Any issues with product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler for Embedded.

In addition to fully supported product features, some product features are only alpha or beta quality.

Beta product features

Beta product features are implementation complete, but have not been sufficiently tested to be regarded as suitable for use in production environments.

Beta product features are identified with [BETA].

- Arm endeavors to document known limitations on beta product features.
- Beta product features are expected to eventually become product features in a future release of Arm Compiler for Embedded 6.
- Arm encourages the use of beta product features, and welcomes feedback on them.
- Any issues with beta product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler for Embedded.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Alpha product features

Alpha product features are not implementation complete, and are subject to change in future releases, therefore the stability level is lower than in beta product features.

Alpha product features are identified with [ALPHA].

- Arm endeavors to document known limitations of alpha product features.
- Arm encourages the use of alpha product features, and welcomes feedback on them.
- Any issues with alpha product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler for Embedded.

Community features

Arm Compiler for Embedded 6 is built on LLVM technology and preserves the functionality of that technology where possible. This means that there are additional features available in Arm Compiler for Embedded that are not listed in the documentation. These additional features are known as community features. For information on these community features, see the Clang Compiler User's Manual.

Where community features are referenced in the documentation, they are identified with [COMMUNITY].

- Arm makes no claims about the quality level or the degree of functionality of these features, except when explicitly stated in this documentation.
- Functionality might change significantly between feature releases.
- Arm makes no guarantees that community features are going to remain functional across update releases, although changes are expected to be unlikely.

Some community features might become product features in the future, but Arm provides no roadmap for this. Arm is interested in understanding your use of these features, and welcomes feedback on them. Arm supports customers using these features on a best-effort basis, unless the features are unsupported. Arm accepts defect reports on these features, but does not guarantee that these issues are going to be fixed in future releases.

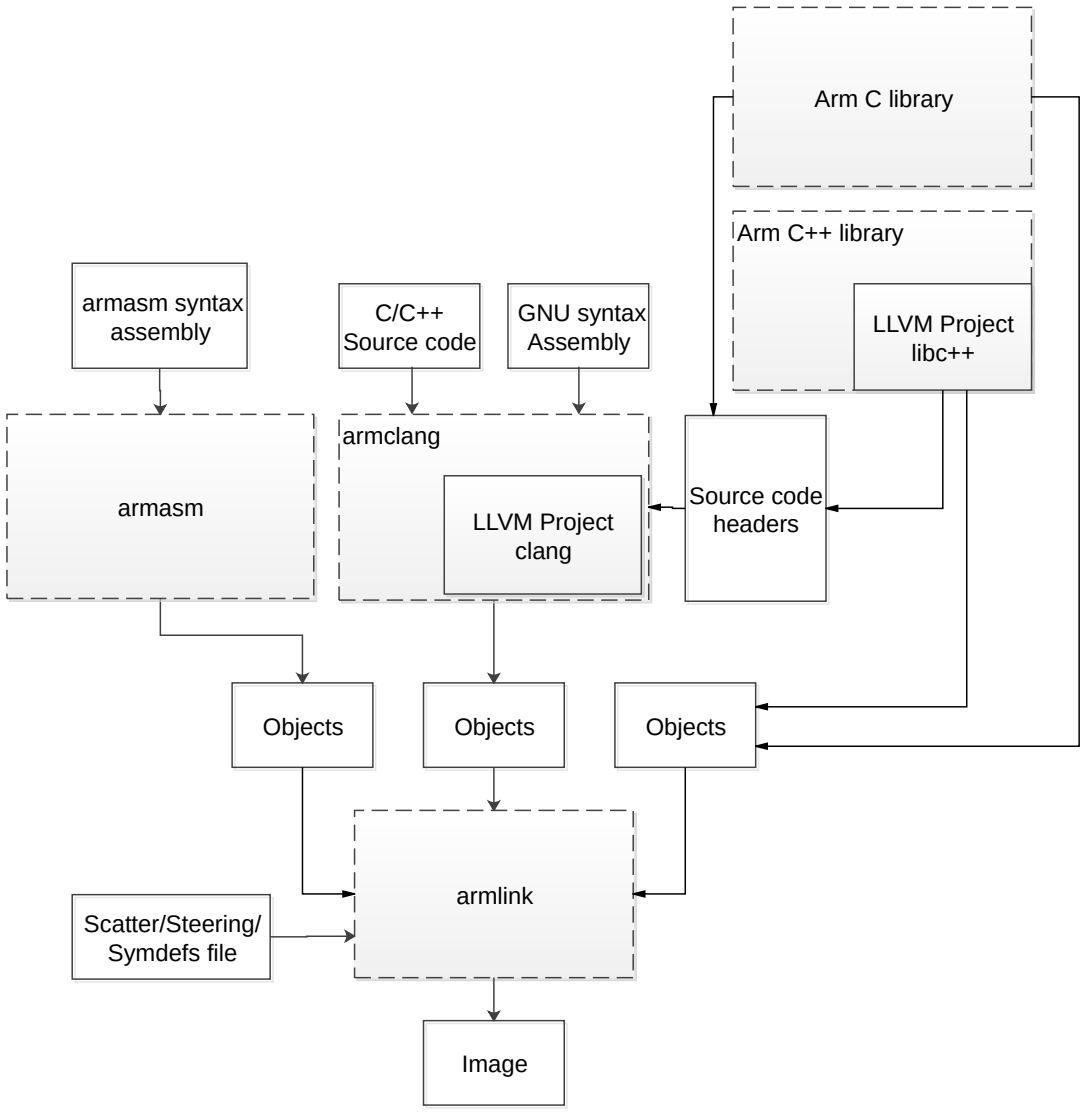
Guidance on use of community features

There are several factors to consider when assessing the likelihood of a community feature being functional:

- The following figure shows the structure of the Arm Compiler for Embedded 6 toolchain:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 1-1: Integration boundaries in Arm Compiler for Embedded 6



The dashed boxes are toolchain components, and any interaction between these components is an integration boundary. Community features that span an integration boundary might have significant limitations in functionality. The exception to this is if the interaction is codified in one of the standards supported by Arm Compiler for Embedded 6. See Application Binary Interface (ABI). Community features that do not span integration boundaries are more likely to work as expected.

- Features primarily used when targeting hosted environments such as Linux or BSD might have significant limitations, or might not be applicable, when targeting bare-metal environments.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The Clang implementations of compiler features, particularly those that have been present for a long time in other toolchains, are likely to be mature. The functionality of new features, such as support for new language features, is likely to be less mature and therefore more likely to have limited functionality.

Deprecated features

A deprecated feature is one that Arm plans to remove from a future release of Arm Compiler for Embedded. Arm does not make any guarantee regarding the testing or maintenance of deprecated features. Therefore, Arm does not recommend using a feature after it is deprecated.

For information on replacing deprecated features with supported features, see the Arm Compiler for Embedded documentation and Release Notes. Where appropriate, each Arm Compiler document includes notes for features that are deprecated, and also provides entries in the changes appendix of that document.

Unsupported features

With both the product and community feature categories, specific features and use cases are known not to function correctly, or are not intended for use with Arm Compiler for Embedded 6.

Limitations of product features are stated in the documentation. Arm cannot provide an exhaustive list of unsupported features or use cases for community features. The known limitations on community features are listed in Community features.

List of known unsupported features

The following is an incomplete list of unsupported features, and might change over time:

- The Clang option `-stdlib=libstdc++` is not supported.
- `-mabi=aapcs-soft` is not supported for A-profile targets in AArch64 state. The `aapcs-soft` ABI is defined only for Armv8-R AArch64 targets. For more information, see the *Soft-float* section of the Procedure Call Standard for the Arm 64-bit Architecture.
- `-mabi=aapcs-soft` is not supported for C++ source language modes.
- C++ static initialization of local variables is not thread-safe when linked against the standard C++ libraries. For thread-safety, you must provide your own implementation of thread-safe functions as described in Standard C++ library implementation definition.


Note

This restriction does not apply to the [ALPHA]-supported multithreaded C++ libraries.

- Use of C11 library features is unsupported.
- Any community feature that is exclusively related to non-Arm architectures is not supported.
- Except for Armv6-M, compilation for targets that implement architectures lower than Armv7 is not supported.
- The `long double` data type is not supported for AArch64 state because of limitations in the current Arm C library.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- C complex arithmetic is not supported, because of limitations in the current Arm C library.
- Complex numbers are defined in C++ as a template, `std::complex`. Arm Compiler for Embedded supports `std::complex` with the `float` and `double` types, but not the `long double` type because of limitations in the current Arm C library.



For C code that uses complex numbers, it is not sufficient to recompile with the C++ compiler to make that code work. How you can use complex numbers depends on whether or not you are building for Armv8-M targets.

- You must take care when mixing translation units that are compiled with and without the [COMMUNITY] `-fsigned-char` 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 supplied with the Arm compilation tools.

- There are limitations with the *Control Flow Integrity* (CFI) sanitizer implementation, `-fsanitize=cfi`, which requires *Link-Time Optimization* (LTO), `-flto`. The following are likely to occur:
  - When using features such as C++ I/O streams, the linker might report errors for a rejected local symbol, `L6654E`, or that a symbol is not preserved by the LTO code generation, `L6137E`.
  - The linker might report a diagnostic that a symbol has a size that extends outside of its containing section, `L6783E` or `L6784E`.

    Use the linker option `--diag_suppress 6783` or `--diag_suppress 6784` to suppress the diagnostic.

Alternatives to C complex numbers not being supported

If you are building for Armv8-M targets, consider using the free and open-source CMSIS-DSP library that includes a data type and library functions for complex number support in C. For more information about CMSIS-DSP and complex number support see the following sections of the CMSIS documentation:

- Complex Math Functions
- Complex Matrix Multiplication
- Complex FFT Functions

If you are not building for Armv8-M targets, consider modifying the affected part of your project to use the C++ standard library type `std::complex` instead.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2. armclang Reference

A list of the command-line options for the `armclang` command, and other reference information that is relevant to `armclang`.

2.1 armclang Command-line Options

`armclang` provides many command-line options, including most Clang command-line options in addition to a number of Arm-specific options.

Additional information about [COMMUNITY] feature command-line options is available in the Clang and LLVM documentation on the LLVM Compiler Infrastructure Project web site.


Note

This topic includes descriptions of [ALPHA] and [COMMUNITY] features. See Support level definitions.


Note

Be aware of the following:

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

Summary of armclang command-line options

The command-line options either affect both compilation and assembly, or only affect compilation. The command-line options that only affect compilation without affecting armclang integrated assembler are shown in the table as *Compilation only*. The command-line options that affect both compilation and assembly are shown in the table as *Compilation and assembly*.


Note

The command-line options that affect assembly are for the `armclang` integrated assembler, and do not apply to `armasm`. These options affect both inline assembly and assembly language source files.


Note

Assembly language source files are assembled using the `armclang` integrated assembler. C and C++ language source files, which can contain inline assembly code, are compiled using the `armclang` compiler. Command-line options that are shown as *Compilation only* do not affect the integrated assembler, but they can affect inline assembly code.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 2-1: armclang command-line options

Option	Description	Compilation or Assembly
-C (armclang)	Keep comments in the preprocessed output.	Compilation and assembly.
-c (armclang)	Only performs the compile step, does not invoke `armlink`.	Compilation and assembly.
-Dname[(parm-list)][=def]	Defines a preprocessor macro.	Compilation and assembly.
-dD, -dM	Prints additional macro information when using `-E`.	Compilation and assembly.
-E	Only performs the preprocess step, does not compile or link.	Compilation and assembly.
-e	Specifies the unique initial entry point of the image.	Compilation and assembly.
-faligned-new	Enables support for types that require more alignment than the aligned C++17 `new` operator provides.	Compilation only.
-fbare-metal-pie	Generates position independent code.	Compilation only.
-fbracket-depth=N	Sets the limit for nested parentheses, brackets, and braces.	Compilation and assembly.
-fcommon, -fno-common	Generates common zero-initialized values for tentative definitions.	Compilation only.
-fcomplete-member-pointers	When the member function pointer call checking scheme, `cfi-mfcall`, is enabled, ensures that `armclang` always emits a full *Control Flow Integrity* (CFI) check.	Compilation only.
-fdata-sections, -fno-data-sections	Enables or disables the generation of one ELF section for each variable in the source file.	Compilation only.
-feliminate-unused-debug-types, -fno-eliminate-unused-debug-types	[COMMUNITY] Specify whether unused types are eliminated from debug information. The `-fno-eliminate-unused-debug-types` option tells the compiler to emit debug symbol output for types that are defined but not used in the source file.	Compilation only.
-fexceptions, -fno-exceptions	Enables or disables the generation of code needed to support C++ exceptions.	Compilation only.
-ffast-math, -fno-fast-math	Enables or disables the use of aggressive floating-point optimizations.	Compilation only.
-ffixed-rN	Prevents the compiler from using the specified core register, unless the use is for Arm ABI compliance.	Compilation only.
-ffixed-x18	Sets the X18 register to the start of the shadow call stack.	Compilation only.
-ffp-contract=value	Specifies when the compiler can perform floating-point expression contractions.	Compilation only.
-ffp-mode=mode	Specifies floating-point standard conformance.	Compilation only.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Option	Description	Compilation or Assembly
-ffreestanding	Asserts that `armclang` must not assume a hosted environment.	Compilation only.
-ffunction-sections, -fno-function-sections	Enables or disables the generation of one ELF section for each function in the source file.	Compilation only.
-fident, -fno-ident	Controls whether the output file contains the compiler name and version information.	Compilation only.
-finstrument-functions	[COMMUNITY] Enables calls to instrumentation functions at function entry and exit.	Compilation only.
file	Reads a list of command-line options from a file.	Compilation and assembly.
-fldm-stm, -fno-ldm-stm	Enables or disables the generation of LDM and STM instructions. AArch32 only.	Compilation only.
-flto, -fno-lto	Enables or disables Link-Time Optimization, and outputs bitcode wrapped in an ELF file for Link-Time Optimization.	Compilation only.
-fno-builtin	Disables generic, non-Arm specific, handling and optimization of standard C and C++ library functions and operators. See also `-nostdlib`.	Compilation only.
-fno-inline-functions	Disables the automatic inlining of functions at optimization levels `-O2` and `-O3`.	Compilation only.
-fomit-frame-pointer, -fno-omit-frame-pointer	Enables or disables the storage of stack frame pointers during function calls.	Compilation only.
-foptimize-sibling-calls, -fno-optimize-sibling-calls	Enables or disables the tail call optimization.	Compilation only.
-fpic, -fno-pic	Enables or disables the generation of position independent code with relative address references, which are independent of the location where your program is loaded.	Compilation only.
-fropi, -fno-ropi	Enables or disables the generation of *Read-Only Position Independent* (ROPI) code.	Compilation only.
-fropi-lowering, -fno-ropi-lowering	Enables or disables runtime static initialization when generating ROPI code.	Compilation only.
-frwpi, -fno-rwpi	Enables or disables the generation of *Read/Write Position Independent* (RWPI) code.	Compilation only.
-frwpi-lowering, -fno-rwpi-lowering	Enables or disables runtime static initialization when generating RWPI code.	Compilation only.
-fsanitize=check[,check,...], -fno-sanitize=check[,check,...]	Enables or disables the sanitizer option used in code generation.	Compilation only.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Option	Description	Compilation or Assembly
-fsanitize-ignorelist, -fno-sanitize-ignorelist	When at least one *Control Flow Integrity* (CFI) sanitizer scheme is enabled with `-fsanitize`, extends or clears the ignore list.	Compilation only.
-fsanitize-minimal-runtime	Enables the minimal handlers mode for *Undefined Behavior Sanitizer* (UBSan) checks.	Compilation only.
-fsanitize-recover=check[,check,...], -fno-sanitize-recover=check[,check,...]	Controls which checks that are enabled by `-fsanitize` are to be non-fatal.	Compilation only.
-fsanitize-trap=check[,check,...], -fno-sanitize-trap=check[,check,...]	Enables or disables the traps mode for UBSan checks.	Compilation only.
-fshort-enums, -fno-short-enums	Allows or disallows the compiler to set the size of an enumeration type to the smallest data type that can hold all enumerator values.	Compilation only.
-fshort-wchar, -fno-short-wchar	Sets the size of `wchar_t` to 2 or 4 bytes.	Compilation only.
-fsigned-char, -funsigned-char	[COMMUNITY] Set the type of `char` to be signed or unsigned (`-funsigned-char` is the default and recommended setting).	Compilation only.
-fsized-deallocation, -fno-sized-deallocation	Enables or disables the C++14 sized deallocation feature.	Compilation only.
-fstack-protector, -fstack-protector-strong, -fstack-protector-all, -fno-stack-protector	Inserts a guard variable onto the stack frame for each vulnerable function or for all functions.	Compilation only.
-fstack-usage	Creates a `.su` file that contains information on the function stack sizes.	Compilation only.
-fstrict-aliasing, -fno-strict-aliasing	Instructs the compiler to apply or not apply the strictest aliasing rules available.	Compilation only.
-fsysv, -fno-sysv	Enables or disables the generation of code suitable for the SysV linking model.	Compilation only.
-ftls-model=model	Specifies the *Thread Local Storage* (TLS) model to be used.	Compilation only.
-ftrapv	Instructs the compiler to generate traps for signed arithmetic overflow on addition, subtraction, and multiplication operations.	Compilation only.
-ftrivial-auto-var-init	Initializes automatic variables to a fixed repeating pattern or to zero.	Compilation only.
-fvectorize, -fno-vectorize	Enables or disables the generation of Advanced SIMD and MVE vector instructions directly from C or C++ code at optimization levels `-O1` and higher.	Compilation only.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Option	Description	Compilation or Assembly
-fvisibility=visibility_type	Sets the default visibility of ELF symbols to the specified option.	Compilation and assembly.
-fwrapv	Instructs the compiler to assume that signed arithmetic overflow of addition, subtraction, and multiplication wraps using two's-complement representation.	Compilation only.
-g, -gdwarf-2, -gdwarf-3, -gdwarf-4, -gdwarf-5	Adds debug tables for source-level debugging.	Compilation and assembly.
-Idir	Adds the specified directory to the list of places that are searched to find include files.	Compilation and assembly.
-include filename	Includes the source code of the specified file at the beginning of the compilation.	Compilation only.
-isystem=directory	Allows you to specify your own directory as a system header directory.	Compilation only.
-L dir [,dir,...]	Specifies a list of paths that the linker searches for user libraries.	Compilation only.
-l name	Adds the specified library to the list of searched libraries.	Compilation only.
-M, -MM	Produces a list of makefile dependency rules suitable for use by a make utility.	Compilation and assembly.
-MD, -MMD	Compiles or assembles source files and produces a list of makefile dependency rules suitable for use by a make utility.	Compilation and assembly.
-MF filename	Specifies a filename for the makefile dependency rules produced by the `-M` and `-MD` options.	Compilation only.
-MG	Prints dependency lines for header files even if the header files are missing.	Compilation only.
-MP	Emits dummy dependency rules that work around make errors that are generated if you remove header files without a corresponding update to the makefile.	Compilation only.
-MT	Changes the target of the makefile dependency rule produced by dependency-generating options.	Compilation and assembly.
-mabi=name	Specifies the calling convention for AArch64 state.	Compilation only.
-march=name[+[no]feature+...]	Targets an architecture profile, generating generic code that runs on any processor of that architecture.	Compilation and assembly.
-marm	Requests that the compiler targets the A32 instruction set.	Compilation only.
-masm=assembler	Selects the correct assembler for the input assembly source files.	Compilation and assembly.
-mbig-endian	Generates code suitable for an Arm processor using byte-invariant big-endian (BE-8) data.	Compilation and assembly.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Option	Description	Compilation or Assembly
-mbranch-protection=protection	Protects branches using Pointer Authentication and Branch Target Identification.	Compilation only.
-mcmodel=model	Selects the generated code model.	Compilation only.
-mcmse	Enables the generation of code for the Secure state of the Armv8-M Security Extension.	Compilation only.
-mcpu=name[+[no]feature+...]	Targets a specific processor, generating optimized code for that specific processor.	Compilation and assembly.
-mdefault-build-attributes, -mno-default-build-attributes	[COMMUNITY] Adds build attributes to files assembled with the integrated assembler.	Assembly only.
-mexecute-only	Generates execute-only code, and prevents the compiler from generating any data accesses to code sections.	Compilation only.
-mfloat-abi=value	Specifies the following: • Whether to use hardware instructions or software library functions for floating-point operations. • Which registers are used to pass floating-point parameters and return values.	Compilation and assembly.
-mfpu=name	Specifies the target FPU architecture, that is the floating-point hardware available on the target.	Compilation and assembly.
-mframe-chain=model	Specifies whether a frame chain is maintained, and the level of compliance of the frame records in the frame chain.	Compilation only.
-mglobal-merge, -mno-global-merge	Enables or disables the merging of global variables.	Compilation only.
-mharden-pac-ret	Enables the generation of code that modifies and hardens return address signing.	Compilation only.
-mimplicit-it=name	Specifies the behavior of the integrated assembler if there are conditional instructions outside `IT` blocks.	Compilation and assembly.
-mlittle-endian	Generates code suitable for an Arm processor using little-endian data.	Compilation and assembly.
-mno-neg-immediates	Disables the substitution of invalid instructions with valid equivalent instructions that use the logical inverse or negative of the specified immediate value.	Compilation and assembly.
-moutline, -mno-outline	Puts identical sequences of code into a separate function.	Compilation only.
-mpixolib	Generates a *Position Independent eXecute Only* (PIXO) library.	Compilation only.
-mpure-code	Alias for `-mexecute-only`.	Compilation only.
-mrestrict-it, -mno-restrict-it	Allows or disallows the generation of complex IT blocks.	Compilation only.
-mthumb	Requests that the compiler targets the T32 instruction set.	Compilation only.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Option	Description	Compilation or Assembly
-mtls-size=size	Controls the maximum size of the offset used when accessing TLS variables.	Compilation only.
-mtp	Sets the exception level for the *Thread Pointer* (TP) register.	Compilation only.
-mtune=target	[COMMUNITY] Allows you to tune the code generation for certain scheduling features and optimizations independently from the architecture. Supported only for AArch64 state.	Compilation only.
-munaligned-access, -mno-unaligned-access	Enables or disables unaligned accesses to data on Arm processors.	Compilation only.
-nobuiltininc	Forces the compiler to exclude the built-in `#include` directories.	Compilation only.
-nostdlib	Forces the compiler to not use the Arm standard C and C++ libraries.	Compilation only.
-nostdlibinc	Forces the compiler to exclude the Arm standard C and C++ library header files.	Compilation only.
-Olevel	Specifies the level of optimization to use when compiling source files.	Compilation only.
-o filename	Specifies the name of the output file.	Compilation and assembly.
-pedantic	Generates warnings if code violates strict ISO C and ISO C++.	Compilation only.
-pedantic-errors	Generates errors if code violates strict ISO C and ISO C++.	Compilation only.
-print-enabled-extensions	Print the extensions enabled by the `-march` or `-mcpu` options for AArch64 state.	Compilation only.
-resource-dir=path_to_resource_folder	Identifies the location of resource files that are used by various `armclang` features.	Compilation only.
-Rpass={*|optimization}	[COMMUNITY] Outputs remarks from the optimization passes made by `armclang`. You can output remarks for all optimizations, or remarks for a specific optimization.	Compilation only.
-S	Outputs the disassembly of the machine code generated by the compiler.	Compilation only.
-save-temps	Instructs the compiler to generate intermediate assembly files from the specified C/C++ file.	Compilation only.
-shared	Creates a System V (SysV) shared object.	Compilation only.
-std=name	Specifies the language standard to compile for.	Compilation only.
--target=triple	Generate code for the specified target triple.	Compilation and assembly.
-U name	Removes any initial definition of the specified preprocessor macro.	Compilation only.
-u symbol	Prevents the removal of a specified symbol if it is undefined.	Compilation and assembly.
-v	Displays the commands that invoke the compiler and sub-tools, such as `armlink`, and executes those commands.	Compilation and assembly.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Option	Description	Compilation or Assembly
--verbose	Displays the same information as `-v`.	Compilation and assembly.
--version	Displays the same information as `--vsn`.	Compilation and assembly.
--version_number	Displays the version of `armclang` you are using.	Compilation and assembly.
--vsn	Displays the version information and the license details.	Compilation and assembly.
-Wname	Controls diagnostics.	Compilation only.
-Wl, opt,[opt[,...]]	Specifies linker command-line options to pass to the linker when a link step is being performed after compilation.	Compilation only.
-Xlinker opt	Specifies linker command-line options to pass to the linker when a link step is being performed after compilation.	Compilation only.
-x language	Specifies the language of source files.	Compilation and assembly.
-###	Displays the commands that invoke the compiler and sub-tools, such as `armlink`, without executing those commands.	Compilation and assembly.

2.1.1 -C (armclang)

Keeps comments in the preprocessed output.

Syntax

```
-C
```

Parameters

None.

Restrictions

You must specify the `-E` option when you use the `-C` option.

Operation

By default, comments are stripped out. Use the `-C` option to keep comments in the preprocessed output.

With the `-C` option, all comments are passed through to the output file, except for comments in processed directives which are deleted along with the directive.

Using the `-C` option does not implicitly select the `-E` option. If you do not specify the `-E` option, the compiler reports:

```
warning: argument unused during compilation: '-C' [-Wunused-command-line-argument]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The -C option can also be used when preprocessing assembly files, using:

- -xassembler-with-cpp, or a file that has an upper-case extension, with the armclang integrated assembler.
- --cpreproc and --cpreproc_opts with the legacy assembler, armasm.

Example: Keep comments in preprocessed output

Here is an example program, foo.c, that contains some comments:

```
#define HIGH 1 // Comment on same line as directive

#define LOW 0
#define LEVEL 10
// #define THIS 99

// Comment A
/* Comment B */

int Signal (int value)
{
  if (value>LEVEL) return HIGH; // Comment C
  return LOW + THIS;
}
```

Use armclang to preprocess this example code with the -C option to retain comments. The -E option executes the preprocessor step only.

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -C -E foo.c
```

The output from the preprocessor contains:

```
// #define THIS 99

// Comment A
/* Comment B */

int Signal (int value)
{
  if (value>10) return 1; // Comment C
  return 0 + THIS;
}
```

The preprocessor has kept all the comments.

The #define directives HIGH, LOW, and LEVEL are converted into their defined values, and the comment alongside HIGH is removed. The #define directive THIS is considered a comment because that line starts with //, and therefore is not converted.

Related information

-E on page 46

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.2 -c (armclang)

Instructs the compiler to perform the compilation step, but not the link step.

Syntax

```
-c
```

Parameters

None.

Operation

We recommend using the `-c` option in projects with more than one source file.

The compiler creates one object file for each source file, with a `.o` file extension replacing the file extension on the input source file.

Example

The following command creates object files `test1.o`, `test2.o`, and `test3.o`:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -c test1.c test2.c test3.c
```

If you specify multiple source files with the `-c` option, the `-o` option results in an error. For example:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -c test1.c test2.c -o
 test.o

armclang: error: cannot specify -o when generating multiple output files
```

2.1.3 -D (armclang)

Defines a macro with a specified name.

Syntax

```
-D<name>[(<parm-list>)][=<def>]
```

Parameters

`<name>`

The name of the macro to be defined.

`<parm-list>`

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.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

Parentheses might require escaping on UNIX systems.

=<def>

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.

Operation

Specifying -D<name> has the same effect as placing the text #define <name> at the head of each source file.

Example: Define a macro to find the maximum of two values

To define a MAX(X,Y) macro, specify the option:

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

This option is equivalent to defining the following macro at the head of each source file:

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

Related information

-include on page 114
-U on page 222
-x (armclang) on page 229
Preprocessing assembly code

2.1.4 -d (armclang)

Prints various states of the preprocessor.

Syntax

Print macro definitions in -E mode in addition to normal output:

```
-dD
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Print macro definitions in `-E` mode instead of normal output:

```
-dM
```

Parameters

None.

Restrictions

You must specify the `-E` option when you use the `-dD` and `-dM` options.

Operation

Using the these options does not implicitly select the `-E` option. However, no warning message is output if you omit `-E`.

Related information

-E on page 46

2.1.5 -E

Executes the preprocessor step only.

Syntax

```
-E
```

Parameters

None.

Operation

By default:

- Output from the preprocessor is sent to the standard output stream and can be redirected to a file using standard UNIX and Windows notation.
- Comments are stripped from the output. Use the `-C` option to keep comments in the preprocessed output.
- Macro definitions are expanded. To obtain a list of the macro definitions that are expanded, use the `-dD` or `-dM` option together with `-E`.

You can also use the `-o` option to specify a file for the preprocessed output.

Preprocessing can also be done for scatter files, as well as for C or C++ source.

Example - expanding macros to produce a simple source file

The following C file `example.c` defines a macro and also includes the header file `example.h`:

```
// example.c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
//
// This incudes a header file and defines a macro
//

#include "example.h"

#define INTA 3

//
// Do a calculation that depends on a value defined in example.h
//
int func(int a)
{
  return a + INTA - INTB;
}
```

The header file defines another macro:

```
// example.h
//
// This defines the INTB value and other values which must not be shared.
//
#define INTB 4
#define SECRET 5 // do not share this value
```

Often a header file might contain a large number of definitions, many of which are not relevant, or are even potentially confidential.

To create a single source file with comments removed, and the macros that are used expanded, use the following command:

```
armclang --target=aarch64-arm-none-eabi -E example.c -o pp_example.c
```

This command results in a single C file which can be understood without needing the separate header file, and with the comments from the original source code stripped out. The simplified C code is as follows:

```
int func(int a)
{
   return a + 3 - 4;
}
```

Example: Generate interleaved macro definitions and preprocessor output

Use `-E -dD` to generate interleaved macro definitions and preprocessor output:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -E -dD source.c > raw.c
```

Example: Create a list of macros

Use `-E -dM` to list all the macros that are defined at the end of the translation unit, including the predefined macros:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -E -dM source.c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Preprocessing a scatter file

Sometimes a scatter file might contain C style macro definitions and a line at the top of the file that tells the linker to preprocess the file before linking. This can result in scatter files that might be hard to understand when sharing as a standalone file.

As with C files, you can use the preprocessor to create a simplified standalone version with comments stripped and macros expanded.

First, make sure you remove any line in the scatter file which is of this form:

```
#! armclang -E <target_triple> -mcpu=<cpu> -xc
```

For example, you might need to remove a line such as this:

```
#! armclang -E --target=arm-arm-none-eabi -mcpu=cortex-m55 -xc
```

This line tells the linker to run the preprocessor before linking. To create a simplified version of the scatter file, you need to remove that line before running the preprocessor direct from the command line.

You can then execute the following command:

```
armclang -E -xc <scatter_file> -o <pp_scatter_file>
```

This command runs the preprocessor on the file and outputs the result in the file specified.

The `-xc` option tells the preprocessor to treat the scatter file as a C file. Alternatively, you could rename the file with a `.c` extension and omit the `-xc` option.

Related information

-c (armclang) on page 43
-d (armclang) on page 45
--target on page 221
-x (armclang) on page 229
Providing source code to Arm support

2.1.6 -e

Specifies the unique initial entry point of the image.

Syntax

```
-e
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

If linking, `armclang` translates this option to `--entry` and passes it to `armlink`. If the link step is not being performed, this option is ignored.

See --entry=location information about the `--entry` linker option.

Related information

Image entry points on page 577

2.1.7 -faligned-new

This option informs the compiler that it can use the overloads of `new` that take a `std::align_val_t` parameter. The overloads were first introduced in C++17.

Default

This option is enabled by default for C++17 and later.

Syntax

```
-faligned-new
```

Parameters

None.

Restrictions

`armclang` does not support the `<size>` parameter for the `-faligned-new` command-line option.

Operation

You must compile with `-faligned-new` when all the following are true:

- The program is compiled in a C++14 or earlier source language mode.
- The program uses `new` to allocate an object or an array of objects `x`.
- The `std::align_val_t` parameter to `new` is not specified.
- `x` has an alignment requirement greater than the value of the predefined macro `__STDCPP_DEFAULT_NEW_ALIGNMENT__`.

Failure to do so can result in the allocated items being underaligned. This situation can subsequently result in unexpected run-time behavior.

If you compile for C++17 (`-std=c++17`) or use `-faligned-new`, then the predefined macro `__cpp_aligned_new` is defined.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Alignment of operator new for C++17

Create the file `test.cpp` containing the following code:

```
struct alignas(32) Test {
  int x;
  int y;
  int z;
};

Test* func(int x1, int y1, int z1) {
  auto ptest = new Test;
  ptest->x = x1;
  ptest->y = y1;
  ptest->z = z1;
  return ptest;
}
```

Compile `test.cpp` with the command for C++17:

```
armclang --target=aarch64-arm-none-eabi -std=c++17 -O0 test.cpp -dM -E
```

The output shows that the predefined macro `__cpp_aligned_new` is defined:

```
...
#define __cpp_aligned_new 201606L
...
```

Because `Test` has an alignment that is larger than `__STDCPP_DEFAULT_NEW_ALIGNMENT__`, `new Test;` calls:

```
void* operator new(std::size_t size, std::align_val_t align)
```

Example: Alignment of operator new for C++14

Compile the program from the previous example for C++14 with and without `-faligned-new` and examine the output from each invocation:

```
armclang --target=aarch64-arm-none-eabi -std=c++14 -O0 test.cpp -dM -E
armclang --target=aarch64-arm-none-eabi -std=c++14 -faligned-new -O0 test.cpp -dM -E
```

With the `-faligned-new` option, the output shows that the predefined macro `__cpp_aligned_new` is defined:

```
...
#define __cpp_aligned_new 201606L
...
```

Related information

-std on page 218
-fno-builtin on page 73
Standard C++ library implementation definition

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

C++ functions you can re-implement
C++ Expressions
Alignment support in Arm Compiler for Embedded 6

2.1.8 -fbare-metal-pie

Generates *Position Independent Executable* (PIE) code.

Syntax

```
-fbare-metal-pie
```

Parameters

None.

Operation

This option causes the compiler to invoke `armlink` with the `--bare_metal_pie` option when performing the link step.

Related information

Support for Position Independent code
Bare-metal Position Independent Executables

2.1.9 -fbracket-depth=N

Sets the limit for nested parentheses, brackets, and braces to `N` in blocks, declarators, expressions, and struct or union declarations.

Default

The default depth limit is 256.

Syntax

```
-fbracket-depth=<N>
```

Parameters

<N>

The depth limit.

Operation

You can increase the depth limit `<N>`.

Related information

Translation limits on page 1045

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.10 -fcommon, -fno-common

Generates common zero-initialized values for tentative definitions. Tentative definitions are declarations of variables with no storage class and no initializer.

Default

The default is `-fno-common`.

Syntax

```
-fcommon
-fno-common
```

Parameters

None.

Operation

The `-fcommon` option places the tentative definitions in a common block. This common definition is not associated with any particular section or object, so multiple definitions resolve to a single symbol definition at link time.

The `-fno-common` option generates individual zero-initialized definitions for tentative definitions. These zero-initialized definitions are placed in a ZI section in the generated object. Multiple definitions of the same symbol in different files can cause a `L6200E: Symbol multiply defined` linker error, because the individual definitions conflict with each other.

The `__attribute__((nocommon))` and `__attribute__((common))` attribute override these command-line options.

Related information

__attribute__((common)), __attribute__((nocommon)) variable attribute on page 295

2.1.11 -fcomplete-member-pointers

When the member function pointer call checking scheme, `cfi-mfcall`, is enabled, it ensures that `armclang` always emits a full *Control Flow Integrity* (CFI) check.

Syntax

```
-fcomplete-member-pointers
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

This option does not modify the code generation, but only enables the compile-time error:

```
... error: member pointer has incomplete base type '<type>'
...
```

Example: Enable all CFI schemes and member function pointer call checking

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 -Omin -flto -fsanitize=cfi
 -fcomplete-member-pointers -fvisibility=hidden ...
```

`-flto` is required when at least one CFI scheme is enabled. `-fvisibility=hidden` is required because the `cfi-derived-cast`, `cfi-unrelated-cast`, `cfi-nvcall`, and `cfi-vcall` schemes are enabled.

Example: Enable a single scheme with member function pointer call checking

The following command enables the `cfi-mfcall` scheme:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 -Omin -flto -fsanitize=cfi-
mfcall -fcomplete-member-pointers ...
```

Related information

-flto, -fno-lto on page 72
-fsanitize, -fno-sanitize on page 84
-fvisibility on page 110
Overview of Control Flow Integrity
Undefined Behavior Sanitizer

2.1.12 -fdata-sections, -fno-data-sections

Enables or disables the generation of one ELF section for each variable in the source file.

Default

The default is `-fdata-sections`.

Syntax

```
-fdata-sections
-fno-data-sections
```

Parameters

None.

Operation

If you want to place specific data items or structures in separate named sections, mark them individually with the `__attribute__((section("<name>")))` variable attribute.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

The `-fdata-sections` option does not disable optimizations such as global merging, `-mglobal-merge`, that might combine variables in a way that affects section placement. An explicit named section `__attribute__((section("<name>")))` disables optimizations that change the name of a section.

Example

If you compare the two Sec columns, you can see that when `-fdata-sections` is used, the variables are put into different sections. When `-fno-data-sections` is used, all the variables are put into the same section.

Create the file `main.c` containing the following C code:

```
volatile int a = 9;
volatile int c = 10;
volatile int d = 11;

int main(void) {
    static volatile int b = 2;
    return a == b;
}
```

Compile this code with:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -fdata-sections -c -O3 main.c
```

Use `fromelf` to see the data in the sections. That is, sections 10, 7, and 8 in the column `Sec`:

```
fromelf -cds main.o

...
    Symbol table .symtab (17 symbols, 11 local)

      #  Symbol Name                 Value      Bind  Sec  Type  Vis  Size
    ============================================================================

     10  .L_MergedGlobals            0x00000000   Lc   10  Data  De   0x8
     11  main.b                      0x00000004   Lc   10  Data  De   0x4
     12  ...
     13  ...
     14  a                           0x00000000   Gb   10  Data  De   0x4
     15  c                           0x00000000   Gb    7  Data  Hi   0x4
     16  d                           0x00000000   Gb    8  Data  Hi   0x4
...
```

Compile the code again with `-fno-data-sections`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -fno-data-sections -c -O3 main.c
```

All symbols `a`, `b`, and `c` are now in Section 7:

```
fromelf -cds main.o

...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    Symbol table .symtab (15 symbols, 10 local)

      #  Symbol Name                 Value      Bind  Sec  Type  Vis  Size
    ==========================================================================

      8  .L_MergedGlobals            0x00000008   Lc    7  Data  De   0x8
      9  main.b                      0x0000000c   Lc    7  Data  De   0x4
     10  ...
     11  ...
     12  a                           0x00000008   Gb    7  Data  De   0x4
     13  c                           0x00000000   Gb    7  Data  Hi   0x4
     14  d                           0x00000004   Gb    7  Data  Hi   0x4
...
```

Related information

-ffunction-sections, -fno-function-sections on page 67
-mglobal-merge, -mno-global-merge on page 175
-O (armclang) on page 203
__attribute__((section("name"))) variable attribute on page 297

2.1.13 -feliminate-unused-debug-types, -fno-eliminate-unused-debug-types

Specify whether debug information is emitted for type symbols that are declared but not used.


Note

This topic describes a [COMMUNITY] feature. See Support level definitions.

Default

The default is `-feliminate-unused-debug-types`.

Syntax

```
-feliminate-unused-debug-types
-fno-eliminate-unused-debug-types
```

Parameters

None.

Operation

By default debug information is only emitted for type symbols that are used somewhere in your program. Type symbols that are declared but never used are eliminated from the debug information.

Specifying `-fno-eliminate-unused-debug-types` tells the compiler to emit debug symbol output for all types that are defined in the source code, even if they are not used.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

-g, -g<level>, -gdwarf-<version> (armclang) on page 111

2.1.14 -fexceptions, -fno-exceptions

Enables or disables the generation of code needed to support C++ exceptions.

Default

The default is `-fexceptions` for C++ sources. The default is `-fno-exceptions` for C sources.

Syntax

```
-fexceptions
-fno-exceptions
```

Parameters

None.

Operation

Compiling with `-fno-exceptions` disables exceptions support and uses the variant of C++ libraries without exceptions. Use of try, catch, or throw results in an error message.

Linking objects that have been compiled with `-fno-exceptions` automatically selects the libraries without exceptions. You can use the linker option `--no_exceptions` to diagnose whether the objects being linked contain exceptions.



Note

If an exception propagates into a function that has been compiled without exceptions support, then the program terminates.

Related information

Standard C++ library implementation definition

2.1.15 -ffast-math, -fno-fast-math

`-ffast-math` tells the compiler to perform more aggressive floating-point optimizations.

Syntax

```
-ffast-math
-fno-fast-math
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

`-ffast-math` 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. Arm recommends that you use the alias option `-ffp-mode=fast` instead of `-ffast-math`.

Using `-fno-fast-math` disables aggressive floating-point optimizations. Arm recommends that you use the alias option `-ffp-mode=full` instead of `-fno-fast-math`.


Note

Arm® Compiler for Embedded 6 does not use `-ffast-math` or `-fno-fast-math` by default. For the default behavior, specify `-ffp-mode=std`.

These options control which floating-point library the compiler uses. For more information, see the C and C++ library naming conventions in *Arm C and C++ Libraries and Floating-Point Support User Guide*.

Table 2-2: Floating-point library variants

armclang option	Floating-point library variant	Description
Default	`fz`	IEEE-compliant except that denormals are flushed to zero and no exceptions are supported.
`-ffast-math`	`fz`	IEEE-compliant with the following exceptions: • Denormals are flushed to zero and no exceptions are supported. • `armclang` performs aggressive floating-point optimizations that might cause a small loss of accuracy.
`-fno-fast-math`	`g`	IEEE-compliant library with configurable rounding mode and support for all IEEE exceptions. This library is a software floating-point implementation, and can result in extra code size and lower performance.


Note

Arm Compiler for Embedded 6 does not support the flushing of denormal half-precision values.

Related information

-ffp-mode on page 62

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.16 -ffixed-r<N>

Prevents the compiler from using the specified general-purpose register, unless the use is required for Arm ABI compliance. The option also prevents the register contents being placed on the stack. If you want to reserve registers for use as a global named register variable, you must use this option.

Default

By default, the compiler is free to use general-purpose registers for any purpose, such as for temporary storage of local variables, within the requirements of the Arm ABI.

Syntax

```
-ffixed-r<N>
```

Parameters

<N>

Specifies the register number from 6-11, and enables you to reserve general-purpose registers R6 to R11.

Restrictions

This feature is only available for AArch32 state.

If you use `-mpixolib`, then you must not use the following registers as global named register variables:

- R8
- R9

If you use `-frwpi` or `-frwpi-lowering`, then you must not use register R9 as a global named register variable. If you do, then `armclang` throws an error.

If needed, the Arm ABI reserves the following registers for use as a frame pointer:

- R7 in T32 state.
- R11 in A32 state.

`armclang` throws an error if you use global named register variables under these conditions.

Code size

Declaring a general-purpose register as a global named register variable means that the register is not available to the compiler for other operations. If you declare too many global named register variables, code size increases significantly. Sometimes, your program might not compile, for example if there are insufficient registers available to compute a particular expression.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

`-ffixed-r<N>` reserves the specified general-purpose register so that the compiler does not use the specified register unless required for Arm ABI compliance. You must reserve the register if you want to use the register as a global named register variable. You can also use `-ffixed-r<N>` for generating compatible objects, for example to generate objects that you want to link with other objects that have been built with `-frwpi`.

For example, `-ffixed-r8` reserves register R8 so that the compiler cannot use R8 for storing temporary variables.


Note

The specified registers might still be used in other object files, for example library code, that have not been compiled using the `-ffixed-r<N>` option.

Examples

The following example demonstrates the effect of the `-ffixed-r<N>` option.

Source file `bar.c` contains the following code:

```
int bar(unsigned int i, unsigned int j,
        unsigned int k, unsigned int l,
        unsigned int m, unsigned int n,
        unsigned int o, unsigned int p,
        unsigned int q, unsigned int r,
        unsigned int s)
{
    return (i + j + k + l + m + n + o + p + q + r + s);
}
```

Compile this code without any `-ffixed-r<N>` option:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O0 -S bar.c -o bar.s
```

The generated assembly file, `bar.s`, saves the registers that it must use, which are `{r4, r5, r6, r7, r8, lr}`:

```
bar:
    ...
    push    {r4, r5, r6, r7, r8, lr}
    ...
    .pad    #16
    sub    sp, sp, #16

    /* Code in between is hidden */

    add    sp, sp, #16
    pop    {r4, r5, r6, r7, r8, pc}
.Lfunc_end0:
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

To ensure that the compiler does not use registers R6, R7, and R8, compile the same code in `foo.c` with the `-ffixed-r6`, `-ffixed-r7`, and `-ffixed-r8` options:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O0 -ffixed-r6 -ffixed-r7 -
ffixed-r8 -S bar.c -o bar.s
```

The generated assembly file, `bar.s`, saves the registers that it must use, which are `{r4, r5, r9, r10, r11, lr}`. In this `bar.s`, the compiler uses registers R9, R10, and R11 instead of R6, R7, and R8:

```
bar:
    ...
    push    {r4, r5, r9, r10, r11, lr}
    ...
    .pad    #16
    sub    sp, sp, #16

    /* Code in between is hidden */

    add    sp, sp, #16
    pop    {r4, r5, r9, r10, r11, pc}
.Lfunc_end0:
```

Related information

Global named register variables on page 252

2.1.17 -ffixed-x18

This option prevents the compiler from using the X18 general-purpose register, unless the use is required for Arm ABI compliance. The option also prevents the register contents being placed on the stack. If you compile with `-fsanitize=shadow-call-stack`, you must use this option.

Default

By default, the compiler is free to use the X18 register for any purpose, such as for temporary storage of local variables.

Syntax

```
-ffixed-x18
```

Parameters

None.

Restrictions

This feature is only available for AArch64 state.

The Arm® Compiler C-library `setjmp()` function saves X18 to `jmp_buf`, and the `longjmp()` function restores X18 from `jmp_buf`. This operation ensures that the use of X18 as a temporary register and the use of X18 by `-fsanitize=shadow-call-stack` work correctly with the `setjmp()` and `longjmp()` functions. If the platform uses X18 to store a program-wide value, such as a pointer to a shared

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

memory buffer, then the program must be aware that `longjmp()` restores the value of X18 to the value when calling `setjmp()`. This restore might cause a problem if the value in X18 has changed since the call to `setjmp()`.

Operation

Use with the shadow call stack sanitizer, `-fsanitize=shadow-call-stack`, to reserve the X18 register to be used by the sanitizer as the shadow stack pointer.


Note

Use of the `-ffixed-x<N>` option for other values of `<N>` is a [COMMUNITY] feature. See *../../arm-compiler-tools-overview/support-level-definitions*.

Related information

-fsanitize, -fno-sanitize on page 84
Parameters in general-purpose registers
longjmp()
setjmp()

2.1.18 -ffp-contract

`-ffp-contract` specifies when the compiler can perform *floating-point* (FP) expression contractions, such that multiple FP operations are combined into one instruction. Combining multiple FP operations can give more precise results because the result is only rounded once after all operations.

For example, use `-ffp-contract=fast` to combine multiply-add operations, if the target has native support for them.

Default

`-ffp-mode`	`-ffp-contract` default
`fast`	`fast`
`std` or `full`	`on`

Syntax

```
-ffp-contract=<value>
```

Parameters

`<value>`

One of the following:

`fast`

Allow FP expression contraction everywhere possible, including across multiple C statements. `#pragma STDC FP_CONTRACT` has no effect on the behavior.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

on

Allow FP expression contraction where possible in the same C statement, unless disallowed by `#pragma STDC FP_CONTRACT`.

off

Do not contract FP expressions, unless allowed by `#pragma STDC FP_CONTRACT`.

Operation

`-ffp-contract=<value>` works by changing the default value of `#pragma STDC FP_CONTRACT <value>` at file-scope. Additionally, the `-ffp-mode=<mode>` option changes the default value of `-ffp-contract=<value>` at file-scope.

Therefore, you can use a combination of `-ffp-contract=<value>` and `#pragma STDC FP_CONTRACT <value>` to configure FP contraction at file-scope and within source files.

`-ffp-contract`	`FP_CONTRACT`	Other options
`on` (Default)	`ON`	The `-ffp-mode=std`, `-ffp-mode=full`, and `-fno-fast-math` options set `-ffp-contract=on`.
`off`	`OFF`	-
`fast`	Always `ON` (`#pragma STDC FP_CONTRACT OFF` has no effect)	The `-ffp-mode=fast` and `-ffast-math` options set `-ffp-contract=fast`.

Example: -ffp-contract examples

The `-ffp-contract` examples show how you can use `-ffp-contract=<value>` and `#pragma STDC FP_CONTRACT <value>` together when compiling with `-ffp-mode=std` to change the behavior of the compiler in different ways.

The examples are available in Examples for the armclang -ffp-contract command-line option.

Related information

-ffp-mode on page 62
#pragma STDC on page 329
Floating-point on page 1051

2.1.19 -ffp-mode

`-ffp-mode` specifies floating-point standard conformance. This option controls the floating-point optimizations that the compiler can perform, and also influences library selection.

Default

The default is `-ffp-mode=std`.

Syntax

```
-ffp-mode=<mode>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<mode>

One of the following:

std

IEEE finite values with denormals flushed to zero, Round to Nearest, and no floating-point exceptions. This mode is compatible with standard C and C++ and is the default option.

Normal finite values are as predicted by the IEEE standard. However, the sign of zero might not be that predicted by the IEEE mode.

The compiler assumes that floating-point exceptions are never raised, and is permitted to transform code under this assumption.


Note

If you are using NaN or infinity, set `-ffp-mode=full`. This option includes arithmetic operations that produce a NaN or infinity, such as when the result of `expf(number)` is larger than the supported maximum value.

If you do not set `-ffp-mode=full`, the compiler outputs a warning:

```
warning: use of NaN/infinity via a macro is undefined behavior
due to the currently enabled floating-point options
```

For `-ffp-mode=std`, the compiler and libraries assume there are no NaNs or infinities, so might produce indeterminate results.

---

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 `__ARM_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.

The compiler assumes that floating-point exceptions are never raised, and is permitted to transform code under this assumption.

Various transformations might be performed, including:

- 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 its reciprocal. For example, `x / 3.0` is evaluated as `x * (1.0 / 3.0)`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- There is no guarantee that the value of `errno` is compliant with the ISO C or C++ standard after math functions are called. This enables the compiler to inline the VFP square root instructions in place of calls to `sqrt()` or `sqrtf()`.


Note

If you are using NaN or infinity, set `-ffp-mode=full`. This option includes arithmetic operations that produce a NaN or infinity, such as when the result of `expf(number)` is larger than the supported maximum value.

If you do not set `-ffp-mode=full`, the compiler outputs a warning:

```
warning: use of NaN/infinity via a macro is undefined behavior
due to the currently enabled floating-point options
```

For `-ffp-mode=fast`, the compiler and libraries assume there are no NaNs or infinities, so might produce indeterminate results.

`full`

All facilities, operations, and representations are available in single and double-precision. Modes of operation can be selected dynamically at runtime.

For AArch64 state, the compiler cannot assume floating-point exceptions are never raised.

The conformance to the floating-point standard from least to most is:

- `fast`
- `std`
- `full`

Restrictions

Floating-point exceptions are supported only for AArch64 state.

Using `_Float16` with `-ffp-mode=full` is not supported.


Note

Arm® Compiler for Embedded 6 does not support the flushing of denormal half-precision values.

Operation

For `std` or `fast` modes, the binary representation of a floating-point number that cannot be represented exactly by its type can be different. The difference depends on whether:

- The compiler evaluates the binary representation at compile time.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The binary representation is generated at runtime using one of the following string to floating-point conversion functions:
  - `atof()`.
  - `strtod()`.
  - `strtof()`.
  - `strtold()`.
  - A member of the `scanf()` family of functions using a floating-point conversion specifier.

To provide more control, `armclang` provides the following symbols to specify the method of binary to decimal conversion at runtime:

- `__use_embedded_btod` - a less accurate conversion that uses less memory, and is more suitable for many embedded applications. Runtime conversions are less accurate than compile-time conversions.
- `__use_accurate_btod` - a more accurate conversion that uses more memory and is slower. Runtime conversions match compile-time conversions.

If you do not explicitly specify one of these symbols, then the default runtime conversion depends on the floating point mode specified using the `-ffp-mode` compiler option:

- `-ffp-mode=std`, `ffp-mode=fast` - embedded quality binary to decimal conversions by default.
- `-ffp-mode=full` - accurate binary to decimal conversions by default.

Floating-point library variant selection

The `std`, `fast`, and `full` options control which floating-point library the compiler uses. For more information, see the C and C++ library naming conventions in the *Arm C and C++ Libraries and Floating-Point Support User Guide*.

Table 2-5: -ffp-mode option and corresponding floating-point library variant

`armclang` option	Floating-point library variant	Description
`-ffp-mode=std`	`fz`	IEEE-compliant except that denormals are flushed to zero and no exceptions are supported.
`-ffp-mode=fast`	`fz`	IEEE-compliant with the following exceptions: • Denormals are flushed to zero and no exceptions are supported. • `armclang` performs aggressive floating-point optimizations that might cause a small loss of accuracy.
`-ffp-mode=full`	`g`	IEEE-compliant library with configurable rounding mode and support for all IEEE exceptions. This library is a software floating-point implementation, and can result in extra code size and lower performance.

Using `-ffp-mode=fast` with *Scalable Vector Extension* (SVE)

The `-ffp-mode=fast` option allows the compiler to use the SVE `FADDV` instruction to perform fast parallel additions across a vector. The `FADDV` instruction is faster than the `FADDA` instruction because `FADDA` performs all additions across the vector in strict sequence.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`math_errhandling` value

A compiler-defined macro, and its meaning is that specified by the C/C++ standards. The value of `math_errhandling` depends on the mode specified for `-ffp-mode` as follows:

Table 2-6: -ffp-mode option and math_errhandling value

`-ffp-mode=<mode>` option	`math_errhandling` value
`fast`	`MATH_ERRNO`
`full`	`MATH_ERRNO | MATH_ERREXCEPT`
`std`	`MATH_ERRNO`

Related information

-ffp-contract on page 61
Library functions on page 1056
C and C++ library naming conventions
A64 -- SVE Instructions (alphabetic order)

2.1.20 -ffreestanding

`-ffreestanding` asserts that `armclang` must not assume a hosted environment, and is typically used to build a C library.


Note

This topic describes a [COMMUNITY] feature. See Support level definitions.

Syntax

```
-ffreestanding
```

Parameters

None.

Operation

The `-ffreestanding` option in Arm® Compiler for Embedded 6 provides no additional implementation for a freestanding environment other than the following:

- Implying the `-fno-builtin` option.
- Defining `__STDC_HOSTED__` to be 0.
- Defining `_LIBCPP_ATOMIC_ONLY_USE_BUILTINS`.

Related information

-fno-builtin on page 73
-std on page 218

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.21 -ffunction-sections, -fno-function-sections

`-ffunction-sections` generates a separate ELF section for each function in the source file. The unused section elimination feature of the linker can then remove unused functions at link time.

Default

The default is `-ffunction-sections`.

Syntax

```
-ffunction-sections
-fno-function-sections
```

Parameters

None.

Post-conditions

`-ffunction-sections` reduces the potential for sharing addresses, data, and string literals between functions. Therefore, there might be a slight increase in code size for some functions.

Operation

The output section for each function has the same name as the function that generates the section, but with a `.text.` prefix. To prevent each function being placed in a separate section, use `-fno-function-sections`.



Note

If you want to place specific data items or structures in separate named sections, mark them individually with `__attribute__((section("<name>")))` variable attribute.

Example

Create the file `functions.c` containing the C code:

```
int function1(int x)
{
  return x+1;
}

int function2(int x)
{
  return x+2;
}
```

Compiling this code with `-ffunction-sections` produces:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -ffunction-sections -S -O3 -o-
 functions.c

...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    .section    .text.function1,"ax",%progbits
    .globl      function1
    .p2align    2
    .type       function1,%function
function1:                              @ @function1
    .fnstart
@ BB#0:
    add         r0, r0, #1
    bx          lr
.Lfunc_end0:
    .size      function1, .Lfunc_end0-function1
    .cantunwind
    .fnend

    .section    .text.function2,"ax",%progbits
    .globl     function2
    .p2align    2
    .type      function2,%function
function2:                              @ @function2
    .fnstart
@ BB#0:
    add        r0, r0, #2
    bx         lr
.Lfunc_end1:
    .size       function2, .Lfunc_end1-function2
    .cantunwind
    .fnend
...
```

Related information

__attribute__((section("name"))) function attribute on page 276
-fdata-sections, -fno-data-sections on page 53
Elimination of unused sections on page 611

2.1.22 -fident, -fno-ident

`-fident` and `-fno-ident` control whether the output file contains the compiler name and version information.

Default

The default is `-fident`.

Syntax

Enable the emission of the compiler name and version information:

```
-fident
```

Alias for `-fident`:

```
-Qy
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Disable the emission of the compiler name and version information:

```
-fno-ident
```

Alias for `-fno-ident`:

```
-Qn
```

Parameters

None.

Operation

The compiler name and version information are output in the following locations:

- The `.ident` directive in assembly files.
- The `.comment` section in object files.
- If debug information is enabled, the `producer` string in debug information.

2.1.23 -finstrument-functions

Inserts instrumentation calls for profiling function entry and exit.


Note

This topic describes a [COMMUNITY] feature. See Support level definitions.

Default

`-finstrument-functions` is disabled by default.

Syntax

```
-finstrument-functions
```

Parameters

None

Operation

Compiling with `-finstrument-functions` causes calls to the following profiling functions to be made on function entry and function exit:

```
void __cyg_profile_func_enter (void *this_fn,
                               void *call_site);
void __cyg_profile_func_exit  (void *this_fn,
                               void *call_site);
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

You must provide implementations of these profiling functions, for example to implement a count of the number of times a function is called. To prevent the profiling functions instrumenting calls to themselves, you can either compile them separately without using `-finstrument-functions`, or you can add `__attribute__((no_instrument_function))` to the definition of the profiling functions.

Related information

__attribute__((no_instrument_function)) function attribute on page 270

2.1.24 @file

Reads a list of `armclang` options from a file.

Syntax

```
@<file>
```

Parameters

`<file>`

The name of a file containing `armclang` options to include on the command line.

Operation

The options in the specified file are inserted in place of the `@<file>` option.

Use whitespace or new lines to separate options in the file. Enclose strings in single or double quotes to treat them as a single word.

You can specify multiple `@<file>` options on the command line to include options from multiple files. Files can contain more `@<file>` options.

If any `@<file>` option specifies a non-existent file or circular dependency, `armclang` exits with an error.



Note

To use Windows-style file paths on the command-line, you must escape the backslashes. For example: `-I"..\\my libs\\"`.

Example

Consider a file `options.txt` with the following content:

```
-I"../my libs/"

--target=aarch64-arm-none-eabi -mcpu=cortex-a57
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Compile a source file `main.c` with the following command line:

```
armclang @options.txt main.c
```

This command is equivalent to the following:

```
armclang -I"../my libs/" --target=aarch64-arm-none-eabi -mcpu=cortex-a57 main.c
```

2.1.25 -fldm-stm, -fno-ldm-stm

Enable or disable the generation of `LDM` and `STM` instructions. AArch32 only.


Note

`-fno-ldm-stm` does not affect the generation of `STRD` and `LDRD` instructions. There is no equivalent option to disable the generation of `STRD` and `LDRD` instructions.

Default

The default is `-fldm-stm`. That is, by default `armclang` can generate `LDM` and `STM` instructions.

Syntax

```
-fldm-stm
-fno-ldm-stm
```

Parameters

None.

Restrictions

Existing `LDM` and `STM` instructions are not removed. For example, instructions in assembly code you are assembling with `armclang`.

`LDM` and `STM` instructions are still present in function prologues and epilogues.

Operation

The `-fno-ldm-stm` option can reduce interrupt latency on systems that:

- Do not have a cache or a write buffer.
- Use zero-wait-state, 32-bit memory.


Note

Using `-fno-ldm-stm` might slightly increase code size and decrease performance.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.26 -flto, -fno-lto

Enables or disables *Link-Time Optimization* (LTO). `-flto` outputs bitcode wrapped in an ELF file for LTO.

Default

The default is `-fno-lto`, except when you specify the optimization levels `-Omax` or `Omin`.

Syntax

```
-flto
-fno-lto
```

Parameters

None.

Restrictions

Object files produced with `-flto` contain bitcode, which cannot be disassembled into meaningful disassembly using the `-s` option or the `fromelf` tool.

Object files generated using the `-flto` option are not suitable for creating ROPI or RWPI images.

LTO performs aggressive optimizations by analyzing the dependencies between bitcode format objects. Such aggressive optimizations can result in the removal of unused variables and functions in the source code.

LTO does not honor the `armclang` option `-mexecute-only`. If you use the `armclang` options `-flto`, `-Omax`, or `-Omin`, then the compiler cannot generate execute-only code.

LTO can interfere with the correct reporting of errors when using file-scope inline assembly. For more information, see File-scope inline assembly.

Operation

If you use `armclang` to invoke the linker, `armclang` automatically passes the `--lto` linker option to `armlink` to enable LTO. For the `-fno-lto` option, `armclang` passes the `--no-lto` linker option to `armlink` to disable LTO.

If you specify the `armclang` option `-c`, you must invoke `armlink` separately with the `--lto` or `--no-lto` option.

`-flto` is automatically enabled when you specify the `armclang` option `-Omax` or `-Omin`.

For more information about LTO, see Optimizing across modules with Link-Time Optimization in the Arm Compiler for Embedded User Guide.

Related information

-c (armclang) on page 43

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--lto, --no_lto

2.1.27 -fno-builtin

Prevents the compiler from recognizing standard library functions as built-in functions, meaning the compiler does not know what the built-in functions do. For example, functions such as `printf()`, `strlen()`, and `malloc()` from the C standard library, or the `new` and `delete` operators from the C++ standard library.


Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.

Default

`-fno-builtin` is disabled by default.

Syntax

```
-fno-builtin
```

Parameters

None.

Operation

A built-in function is a function for which the compiler knows something about the behavior, but not necessarily everything. A built-in function often means a function that is handled completely by the compiler. That is, the compiler converts the function call in the source code into code in the output that does not include a function call. For example, all `__builtin_<name>` functions or various ACLE intrinsics. Also, by default, the standard library functions such as `memcpy` are built-in functions.

`-fno-builtin` influences how the compiler interprets the source code that it sees. The option does not restrict what code the compiler is allowed to generate.

When compiling without `-fno-builtin`, the compiler can replace calls to certain standard library functions with inline code or with calls to other library functions. The Run-time ABI for the Arm Architecture lists library functions that the compiler can use. This means that your reimplementations of the standard library functions might not be used, and might be removed by the linker.


Note

The `-fno-builtin` option does not prevent Arm-specific transformations from within the Arm implementation of the C standard library. Use the `-nostdlib` option to tell the compiler not to use the Arm standard C and C++ libraries.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Compile without -fno-builtin for memcpy

The following test program, `memcpytest.c`, uses the `memcpy()` function:

```
// memcpytest.c
#include <string.h>
void fn1(char *p, char *q) {
  memcpy(p, q, 128);
}
struct example {char arr[128]; };
void fn2(struct example *p, struct example *q) {
  *p = *q;
}
```

Compile `memcpytest.c` without `-fno-builtin`:

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -S -o memcpytest.s
 memcpytest.c
```

With the default, `memcpytest.s` shows that both functions result in a call to `__aeabi_memcpy` as follows:

```
...
fn1:
...
    str     r0, [sp, #4]
    str     r1, [sp]
    ldr     r0, [sp, #4]
    ldr     r1, [sp]
    mov     r2, #128
    bl      __aeabi_memcpy
    add     sp, sp, #8
    pop     {r11, pc}
...
fn2:
...
    str     r0, [sp, #4]
    str     r1, [sp]
    ldr     r0, [sp, #4]
    ldr     r1, [sp]
    mov     r2, #128
    bl      __aeabi_memcpy
    add     sp, sp, #8
    pop     {r11, pc}
...
```

Example: Compile with -fno-builtin for memcpy

Compile `memcpytest.c` with `-fno-builtin`:

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -fno-builtin -S -o
 memcpytest.s memcpytest.c
```

`memcpytest.s` shows that compiling with `-fno-builtin` does not generate the call to `__aeabi_memcpy` in `fn1` because the compiler does not know how `memcpy` works. However, `fn2` generates the call to `__aeabi_memcpy` as follows:

```
...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
fn1:
...
    str    r0, [sp, #4]
    str    r1, [sp]
    ldr    r0, [sp, #4]
    ldr    r1, [sp]
    mov    r2, #128
    bl     memcpy
    add    sp, sp, #8
    pop    {r11, pc}
...
fn2:
...
    str    r0, [sp, #4]
    str    r1, [sp]
    ldr    r0, [sp, #4]
    ldr    r1, [sp]
    mov    r2, #128
    bl     __aeabi_memcpy
    add    sp, sp, #8
    pop    {r11, pc}
...
```

You can use the [COMMUNITY] option `-ffreestanding` instead of `-fno-builtin`. `-ffreestanding` implies `-fno-builtin`, and indicates to the compiler that you are not using a standard library. The compiler assumes that the runtime ABI functions are available even with `-ffreestanding`. Therefore, the compiler still generates a call to `__aeabi_memcpy` in `fn2`.

Example: Compile without -fno-builtin for printf

The following test program, `printftest.c`, uses the `printf()` function:

```
// printftest.c
#include "stdio.h"

void foo( void )
{
    printf("Hello\n");
}
```

Compile `printftest.c` without `-fno-builtin`:

```
armclang -c -O2 -g --target=arm-arm-none-eabi -mcpu=cortex-a9 -mfpu=none -S
 printftest.c -o printftest.s
```

```
...
printftest:
.Lfunc_begin0:
...
    adr    r0, .LCPI0_0
.Ltmp0:
    b    puts
.Ltmp1:
...
```

The compiler has replaced the `printf()` function with the `puts()` function.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Compile with -fno-builtin for printf

Compile `printftest.c` with `-fno-builtin`:

```
armclang -c -O2 -g --target=arm-arm-none-eabi -mcpu=cortex-a9 -mfpu=none -S -fno-
builtin printftest.c -o printftest.s
```

```
printftest:
.Lfunc_begin0:
...
    adr    r0, .LCPI0_0
.Ltmp0:
    b    __2printf
...
```

The compiler has not replaced the `printf()` function with the `puts()` function when using the `-fno-builtin` option. Instead, it has replaced `printf()` with `__2printf` from the Arm standard C library. To prevent this happening, you must also use the `-nostdlib` option.

Example: Compile with -fno-builtin and -nostdlib for printf

Compile `printftest.c` with `-fno-builtin` and `-nostdlib`:

```
armclang -c -O2 -g --target=arm-arm-none-eabi -mcpu=cortex-a9 -mfpu=none -S -fno-
builtin -nostdlib printftest.c -o printftest.s
```

```
...
printftest:
...
        adr    r0, .LCPI0_0
.Ltmp0:
        b    printf
...
```

The compiler has not replaced the `printf()` function.

Related information

-ffreestanding on page 66
-nostdlib on page 200
-nostdlibinc on page 201
Avoid linking in the Arm C library
Run-time ABI for the Arm Architecture

2.1.28 -fno-inline-functions

Disabling the inlining of functions can help to improve the debug experience.

Syntax

```
-fno-inline-functions
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

Although the compiler attempts to automatically inline functions at all optimization levels except at `-O0`, the threshold at which it decides to inline depends on the level. For more information, see Selecting optimization options for more information.

When using `-fno-inline-functions`, no automatic inlining is attempted, and only functions that are tagged with `__attribute__((always_inline))` are inlined.

Related information

-O (armclang) on page 203
__attribute__((always_inline)) function attribute on page 259
Inline functions on page 341
Inlining functions

2.1.29 -fomit-frame-pointer, -fno-omit-frame-pointer

`-fomit-frame-pointer` omits the storing of stack frame pointers during function calls.

Default

The default is `-fomit-frame-pointer`.

Syntax

```
-fomit-frame-pointer
-fno-omit-frame-pointer
```

Parameters

None.

Operation

The `-fomit-frame-pointer` option instructs the compiler to not store stack frame pointers if the function does not need it. You can use this option to reduce the code image size.

The `-fno-omit-frame-pointer` option instructs the compiler to store the stack frame pointer in a register. In AArch32, the frame pointer is stored in register `R11` for A32 code or register `R7` for T32 code. In AArch64, the frame pointer is stored in register `X29`. The register that is used as a frame pointer is not available for use as a general-purpose register. It is available as a general-purpose register if you compile with `-fomit-frame-pointer`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

We do not recommend using the `-fno-omit-frame-pointer` option if you use the `armlink` options `--callgraph`, `--info=stack`, or `--info=summarystack` to measure the stack usage of the function. Local variables on the stack that are accessed through the frame pointer are not counted in the size of the stack frame. Therefore, `armlink` underestimates the amount of stack used.

Frame pointer limitations for stack unwinding

Frame pointers enable the compiler to insert code to remove the automatic variables from the stack when C++ exceptions are thrown. This is called stack unwinding. However, there are limitations on how the frame pointers are used:

- By default, there are no guarantees on the use of the frame pointers.
- There are no guarantees about the use of frame pointers in the C or C++ libraries.
- If you specify `-fno-omit-frame-pointer`, then any function which uses space on the stack creates a frame record, and changes the frame pointer to point to it. There is a short time period at the beginning and end of a function where the frame pointer points to the frame record in the caller's frame.
- If you specify `-fno-omit-frame-pointer`, use `-mframe-chain` to enable the frame chain model and set the level of compliance of the frame records in the frame chain. For more information about `-mframe-chain` see -mframe-chain.
- If you specify `-fno-omit-frame-pointer`, then the frame pointer always points to the lowest address of a valid frame record. A frame record consists of two words:
  - The value of the frame pointer at function entry in the lower-addressed word.
  - The value of the link register at function entry in the higher-addressed word.
- A function that does not use any stack space does not need to create a frame record, and leaves the frame pointer pointing to the caller's frame.
- In AArch32 state, there is currently no reliable way to unwind mixed A32 and T32 code using frame pointers.

Related information

-mframe-chain on page 173
--callgraph, --no_callgraph on page 434
--info=topic[,topic,...] (armlink) on page 474

2.1.30 -foptimize-sibling-calls, -fno-optimize-sibling-calls

Enables or disables tail call optimization.

Default

The default is `-foptimize-sibling-calls`.

Syntax

```
-foptimize-sibling-calls
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
-fno-optimize-sibling-calls
```

Parameters

None.

Example: Disabling tail call optimization

Create the file `test.c` containing the following code:

```
extern int func(int a);

int test(int a)
{
    return func(a);
}
```

Compile `test.c` without `-fno-optimize-sibling-calls`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -O1 -S test.c -o test.s
```

The assembler output contains:

```
    ...
test:
    b   func
    ...
```

Compile `test.c` with `-fno-optimize-sibling-calls`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -O1 -fno-optimize-sibling-
calls -S test.c -o test.s
```

The assembler output now contains:

```
    ...
test:
    str x30, [sp, #-16]!        // 8-byte Folded Spill
    bl  func
    ldr x30, [sp], #16          // 8-byte Folded Reload
    ret
    ...
```

2.1.31 -fpic, -fno-pic

Enables or disables the generation of position independent code with relative address references, which are independent of the location where your program is loaded.

Default

The default is `-fno-pic`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
-fpic
-fno-pic
```

Parameters

None.

Operation

If you use `-fpic`, then 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.

Position independent code compiled with `-fpic` is suitable for use in SysV shared objects.

`-fpic` causes the compiler to invoke `armlink` with the `--fpic` option when performing the link step.


Note

When building a shared library, use `-fpic` together with either the `-fvisibility` option or the visibility attribute, to control external visibility of functions and variables.

Related information

-ftls-model on page 105

2.1.32 -fropi, -fno-ropi

Enables or disables the generation of Read-Only Position Independent (ROPI) code.

Default

The default is `-fno-ropi`.

Syntax

```
-fropi
-fno-ropi
```

Parameters

None.

Restrictions

The following restrictions apply:

- This option is not supported in AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- This option cannot be used with C++ code.
- This option is not compatible with `-fpic`, `-fpie`, or `-fbare-metal-pie` options.

Operation

When generating ROPI code, the compiler:

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


Note

- This option is independent from `-frwpi`, meaning that these two options can be used individually or together.
- When using `-fropi`, `-fropi-lowering` is automatically enabled.

Related information

-frwpi, -fno-rwpi on page 82
-frwpi-lowering, -fno-rwpi-lowering on page 83
-fropi-lowering, -fno-ropi-lowering on page 81

2.1.33 -fropi-lowering, -fno-ropi-lowering

Enables or disables runtime static initialization when generating Read-Only Position Independent (ROPI) code.

Default

The default is `-fno-ropi-lowering`.

If `-fropi` is used, then the default is `-fropi-lowering`.

If `-frwpi` is used without `-fropi`, then the default is `-fropi-lowering`.

Syntax

```
-fropi-lowering
-fno-ropi-lowering
```

Parameters

None.

Operation

If you compile with `-fropi-lowering`, then the static initialization is done at runtime. It is done by the same mechanism that is used to call the constructors of static C++ objects that must run before `main()`. This mechanism enables these static initializations to work with ROPI code.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.34 -frtti, -fno-rtti

Controls the support for the *Run-Time Type Information* (RTTI) features `dynamic_cast` and `typeid` in C++.

Default

The default is `-frtti`.

Syntax

```
-frtti
-fno-rtti
```

Parameters

None.

Restrictions

`-fno-rtti` is ignored if you specify `-fexceptions`, because C++ exceptions require RTTI.

Operation

If you use `-fno-rtti`, `armclang`:

- Outputs an error, if your code includes `typeid`:

  ```
  error: use of typeid requires -frtti
  ```

  However, an error is output for `dynamic_cast` only if the way it is used requires RTTI.
- Does not generate RTTI tables in the object file, or reference the RTTI for basic types as defined in libc++abi.

Related information

About Run-Time Type Information
Avoid linking in Run-Time Type Information

2.1.35 -frwpi, -fno-rwpi

Enables or disables the generation of *Read/Write Position Independent* (RWPI) code.

Default

The default is `-fno-rwpi`.

Syntax

```
-frwpi
-fno-rwpi
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Restrictions

The following restrictions apply:

- This option is not supported in AArch64 state.
- This option is not compatible with `-fpic`, `-fpie`, or `-fbare-metal-pie` options.

Operation

When generating RWPI code, the compiler:

- Addresses the 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

- This option is independent from `-fropi`, meaning that these two options can be used individually or together.
- When using `-frwpi`, `-frwpi-lowering` and `-fropi-lowering` are automatically enabled.

Related information

-fropi, -fno-ropi on page 80
-fropi-lowering, -fno-ropi-lowering on page 81
-frwpi-lowering, -fno-rwpi-lowering on page 83

2.1.36 -frwpi-lowering, -fno-rwpi-lowering

Enables or disables runtime static initialization when generating *Read/Write Position Independent* (RWPI) code.

Default

The default is `-fno-rwpi-lowering`.

If `-frwpi` is used, then the default is `-frwpi-lowering`.

Syntax

```
-frwpi-lowering
-fno-rwpi-lowering
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

If you compile with `-frwpi-lowering`, then the static initialization is done at runtime by the C++ constructor mechanism for both C and C++ code. This mecahnism enables these static initializations to work with RWPI code.

2.1.37 -fsanitize, -fno-sanitize

`-fsanitize` enables checks for various forms of behavior that are intended to damage the integrity of the program execution environment. You can disable a specific check with `-fno-sanitize`.



Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.

Default

The default is no sanitizers are selected.

Syntax

```
-fsanitize=<option>[,<option>,...]
-fno-sanitize=<option>[,<option>,...]
```

Parameters

`<option>`

Specifies the runtime checks to enable:

`bounds`

[COMMUNITY] This sanitizer checks for out of bounds array indexing, in cases where the array bound can be statically determined. The check also includes `-fsanitize=array-bounds` and `-fsanitize=local-bounds`. However, `-fsanitize=local-bounds` is not included in `-fsanitize=undefined`.

`cfi` or a specific scheme

Specify `cfi` to enable all schemes, or enable one or more *Control Flow Integrity* (CFI) sanitizer schemes listed in the following table:

Table 2-7: Control Flow Integrity schemes supported

Scheme	Description
`cfi-cast-strict`	Enables strict cast checks.
`cfi-derived-cast`	Base-to-derived cast to the wrong dynamic type.
`cfi-unrelated-cast`	Cast from `void*` or another unrelated type to the wrong dynamic type.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Scheme	Description
`cfi-nvcall`	Non-virtual call through an object that has a `vptr` of the wrong dynamic type.
`cfi-vcall`	Virtual call through an object that has a `vptr` of the wrong dynamic type.
`cfi-icall`	Indirect call of a function with wrong dynamic type.
`cfi-mfcall`	Indirect call through a member function pointer with wrong dynamic type.

`function`

The function sanitizer is part of the undefined behavior sanitizer and is enabled by `-fsanitize=undefined`. It protects against accidentally calling a function through a function pointer when that pointer does not match the type of function being called.

If an indirect function call in one object can possibly reach a function in another, then those two objects must agree on whether the function sanitizer is enabled. That is, they both must either be compiled with `-fsanitize=function` or without. Otherwise, the generated code might result in unexpected runtime behavior.

`memtag-heap`

Memory tagging sanitizer. Enables the generation of memory tagging code for protecting the memory allocations on the heap.

`memtag-stack`

Memory tagging sanitizer. Enables the generation of memory tagging code for protecting the memory allocations on the stack.

`shadow-call-stack`

Supported for AArch64 state only.

Compiling with shadow call stack sanitizer enabled does not protect the Arm C/C++ libraries because they do not have a variant with shadow call stack sanitizer support.

This sanitizer writes the return address to a shadow call stack, and to the current active stack. On function exit, the return address is loaded from the shadow call stack instead of from the current active stack.

You must compile your program with `-ffixed-x18`, including any libraries, and the runtime must create a shadow stack and set the X18 register to the start of that stack. See -ffixed-x18 for more information.


Note

The shadow call stack grows upwards (ascending), whereas the regular stack grows downwards (full descending).

For more information, see Shadow Call Stack.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`undefined` or a specific UBSan check

Undefined Behavior Sanitizer (UBSan) checks. The complete list of checks are listed under *Available checks* at Undefined Behavior Sanitizer.

You can use a comma-separated list to specify multiple runtime checks.

Restrictions

Control Flow Integrity

Arm® Compiler for Embedded does not support the `-flto=thin` and `-fno-sanitize-trap` options.

All CFI schemes require *Link-Time Optimization* (LTO). If you enable at least one CFI scheme, then you must also compile with the `armclang` option `-flto` and link with the `armlink` option `--lto`.

Note

The `cfi-derived-cast`, `cfi-unrelated-cast`, `cfi-nvcall`, and `cfi-vcall` schemes require that you also specify a `-fvisibility` option. This option is required because the the default setting, `-fvisibility=default`, disables the CFI checks for classes without visibility attributes. Typically, you would want to specify `-fvisibility=hidden`, which enables CFI checks for such classes.

Memory tagging

When compiling with `memtag-heap` or `memtag-stack`, the compiler uses memory tagging instructions that are not available for architectures without the Memory Tagging Extension. The resulting code cannot execute on architectures without the Memory Tagging Extension. For more information, see the `+memtag` feature in -mcpu.

Note

Arm Compiler for Embedded does not support `-fsanitize=memtag`. `-fsanitize=memtag` enables all the memtag sanitizers:

- `-fsanitize=memtag-heap`
- `-fsanitize=memtag-stack`
- `-fsanitize=memtag-globals`

Arm Compiler for Embedded supports `-fsanitize=memtag-heap` and `-fsanitize=memtag-stack`, but does not support `-fsanitize=memtag-globals`. If your source file includes a global variable and you compile with the `-fsanitize=memtag-globals` option, the compiler reports the following error:

```
error: Tagged symbols (-fsanitize=memtag-globals) are only supported on aarch64 + Android
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Function

The function sanitizer stores a piece of data immediately before the start of each function. This data indicates the type of the function. Before calling the function through a function pointer, the sanitizer loads that data and checks that it matches the type of the pointer.

If you compile with the `-mexecute-only` option, the function sanitizer cannot work, because that option prevents the loading of data from your code segment. For that reason, `-fsanitize=function` is incompatible with `-mexecute-only`.

Example: -fsanitize examples

The `-fsanitize` examples show the use of the `memtag-stack` and `function` sanitizers.

The examples are available in Examples for the armclang -fsanitize command-line option.

Related information

-fcomplete-member-pointers on page 52
-ffixed-x18 on page 60
-flto, -fno-lto on page 72
-fsanitize-ignorelist, -fno-sanitize-ignorelist on page 87
-fsanitize-minimal-runtime on page 89
-fsanitize-recover, -fno-sanitize-recover on page 93
-fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector on page 100
-resource-dir on page 212
__attribute__((no_sanitize("option"))) function attribute on page 271
--lto, --no_lto on page 498
Overview of Control Flow Integrity
Overview of memory tagging
Overview of Undefined Behavior Sanitizer
Shadow Call Stack
Undefined Behavior Sanitizer

2.1.38 -fsanitize-ignorelist, -fno-sanitize-ignorelist

Enables you to extend or clear the ignore list used by supported sanitizers. To extend the ignore list using multiple ignore list files, specify a separate `-fsanitize-ignorelist` option for each file.

Syntax

```
-fsanitize-ignorelist=<ignorelistfile>
-fno-sanitize-ignorelist
```

Parameters

`<ignorelistfile>`

A text file containing a list of entities for supported sanitizers. These entities are:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Control Flow Integrity (CFI)

When at least one CFI sanitizer scheme is enabled with `-fsanitize`, extends or clears the ignore list. The supported entities for which the CFI checks are to be relaxed:

Table 2-8: CFI ignore list entity types

Entity	Example

Source files, `src`	`src:bad_file.cpp`
	
	`src:bad_file.h`

Functions, `fun`	`fun:*Foo*`
Types, `typ`	`typ:std::*`

Undefined Behavior Sanitizer (UBSan)

When at least one UBSan sanitizer check is enabled with `-fsanitize`, extends or clears the ignore list. The supported entities for which the UBSan check can suppress error reports:

Table 2-9: UBSan ignore list entity types

Entity	Example
Source files, `src`	`src:/path/to/source/file.c`
Functions, `fun`	`fun:*Foo*`

Operation

`-fno-sanitize-ignorelist` instructs `armclang` to ignore the default `cfi_ignorelist.txt` file.

If you specify a CFI or UBSan check with `-fsanitize` without specifying an ignore list file with `-fsanitize-ignorelist`, then `armclang` uses the default `cfi_ignorelist.txt` file to set up the ignore list. By default, `armclang` searches for this file in `<install_path>/lib/clang/<version>/share`.

Example: Command specifying two ignore list files

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 -fsanitize-
ignorelist=ignorefile1.txt -fsanitize-ignorelist=ignorefile2.txt ...
```

Example: Ignore list file specifying one or more CFI schemes using the [<scheme>] header

```
# Disable CFI call checks for this function without affecting cast checks
[cfi-vcall|cfi-nvcall|cfi-icall]
fun:*BadCall*
```

Related information

-fsanitize, -fno-sanitize on page 84
Overview of Control Flow Integrity
Undefined Behavior Sanitizer

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.39 -fsanitize-minimal-runtime

Enables the minimal handlers mode. In this mode, on detection of undefined behaviors, the program makes calls to specific handlers based on the class of the error. You must implement these handlers in a separate runtime library.

Syntax

```
-fsanitize-minimal-runtime
```

Parameters

None.

Restrictions

`-fsanitize-minimal-runtime` does not support `-fsanitize=function` and `-fsanitize=vptr` checking.

Operation

The prototype for the runtime handler functions is:

```
void <handler_function>()
```

The `<handler_function>` names for recovery and non-recovery actions are shown in the following table:

Table 2-10: Minimal runtime recovery and non-recovery function names

Handler name	Recovery function name	Non-recovery function name
add_overflow	__ubsan_handle_add_overflow_minimal	__ubsan_handle_add_overflow_minimal_abort
alignment_assumption	__ubsan_handle_alignment_assumption_minimal	__ubsan_handle_alignment_assumption_minimal_abort
builtin_unreachable	-	__ubsan_handle_builtin_unreachable_minimal_abort
cfi_check_fail	__ubsan_handle_cfi_check_fail_minimal	__ubsan_handle_cfi_check_fail_minimal_abort
divrem_overflow	__ubsan_handle_divrem_overflow_minimal	__ubsan_handle_divrem_overflow_minimal_abort
float_cast_overflow	__ubsan_handle_float_cast_overflow_minimal	__ubsan_handle_float_cast_overflow_minimal_abort
function_type_mismatch	__ubsan_handle_function_type_mismatch_minimal	__ubsan_handle_function_type_mismatch_minimal_abort
implicit_conversion	__ubsan_handle_implicit_conversion_minimal	__ubsan_handle_implicit_conversion_minimal_abort
invalid_builtin	__ubsan_handle_invalid_builtin_minimal	__ubsan_handle_invalid_builtin_minimal_abort

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Handler name	Recovery function name	Non-recovery function name
invalid_objc_cast	__ubsan_handle_invalid_objc_cast_minimal	__ubsan_handle_invalid_objc_cast_minimal_abort
load_invalid_value	__ubsan_handle_load_invalid_value_minimal	__ubsan_handle_load_invalid_value_minimal_abort
missing_return	-	__ubsan_handle_missing_return_minimal_abort
mul_overflow	__ubsan_handle_mul_overflow_minimal	__ubsan_handle_mul_overflow_minimal_abort
negate_overflow	__ubsan_handle_negate_overflow_minimal	__ubsan_handle_negate_overflow_minimal_abort
nonnull_arg	__ubsan_handle_nonnull_arg_minimal	__ubsan_handle_nonnull_arg_minimal_abort
nonnull_return	__ubsan_handle_nonnull_return_minimal	__ubsan_handle_nonnull_return_minimal_abort
nullability_arg	__ubsan_handle_nullability_arg_minimal	__ubsan_handle_nullability_arg_minimal_abort
nullability_return	__ubsan_handle_nullability_return_minimal	__ubsan_handle_nullability_return_minimal_abort
out_of_bounds	__ubsan_handle_out_of_bounds_minimal	__ubsan_handle_out_of_bounds_minimal_abort
pointer_overflow	__ubsan_handle_pointer_overflow_minimal	__ubsan_handle_pointer_overflow_minimal_abort
shift_out_of_bounds	__ubsan_handle_shift_out_of_bounds_minimal	__ubsan_handle_shift_out_of_bounds_minimal_abort
sub_overflow	__ubsan_handle_sub_overflow_minimal	__ubsan_handle_sub_overflow_minimal_abort
type_mismatch	__ubsan_handle_type_mismatch_minimal	__ubsan_handle_type_mismatch_minimal_abort
vla_bound_not_positive	__ubsan_handle_vla_bound_not_positive_minimal	__ubsan_handle_vla_bound_not_positive_minimal_abort

Example: Compile with a custom library with minimal runtime capabilities

The following code, `libubsan.c`, is a possible implementation:

```
#include <stdio.h>
#include <stdlib.h>

static void message(const char *msg)
{
    puts(msg);
}

static void message_with_abort(const char *msg)
{
    puts(msg);
    abort();
}

#define HANDLER_RECOVER(name, msg)                                  \
  void __ubsan_handle_##name##_minimal() {                          \
    message("Undefined behavior detected:\n" msg);  \
  }
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
#define HANDLER_NORECOVER(name, msg)                                  \
  void __ubsan_handle_##name##_minimal_abort() {                      \
    message_with_abort("Undefined behavior detected:\n" msg); \
  }

#define HANDLER(name, msg)                                            \
    HANDLER_RECOVER(name, msg)                                        \
    HANDLER_NORECOVER(name, msg)                                      \

HANDLER(type_mismatch, "type-mismatch")
HANDLER(alignment_assumption, "alignment-assumption")
HANDLER(add_overflow, "add-overflow")
HANDLER(sub_overflow, "sub-overflow")
HANDLER(mul_overflow, "mul-overflow")
HANDLER(negate_overflow, "negate-overflow")
HANDLER(divrem_overflow, "divrem-overflow")
HANDLER(shift_out_of_bounds, "shift-out-of-bounds")
HANDLER(out_of_bounds, "out-of-bounds")
HANDLER_RECOVER(builtin_unreachable, "builtin-unreachable")
HANDLER_RECOVER(missing_return, "missing-return")
HANDLER(vla_bound_not_positive, "vla-bound-not-positive")
HANDLER(float_cast_overflow, "float-cast-overflow")
HANDLER(load_invalid_value, "load-invalid-value")
HANDLER(invalid_builtin, "invalid-builtin")
HANDLER(invalid_objc_cast, "invalid-objc-cast")
HANDLER(function_type_mismatch, "function-type-mismatch")
HANDLER(implicit_conversion, "implicit-conversion")
HANDLER(nonnull_arg, "nonnull-arg")
HANDLER(nonnull_return, "nonnull-return")
HANDLER(nullability_arg, "nullability-arg")
HANDLER(nullability_return, "nullability-return")
HANDLER(pointer_overflow, "pointer-overflow")
HANDLER(cfi_check_fail, "cfi-check-fail")
```

Create the file `bootcode.s` containing:

```
    .section StartUp, "ax"
    .balign 0x20
    .global Vectors

Vectors:
    b    Reset_Handler
    b    Undefined_Handler
    b    SVC_Handler
    b    Prefetch_Handler
    b    Abort_Handler
    b .  //Reserved vector
    b    IRQ_Handler
    b    FIQ_Handler
// -----------------------------------------------------------------
// Handlers for unused exceptions
// -----------------------------------------------------------------

Undefined_Handler:
    b    Undefined_Handler
SVC_Handler:
    b    SVC_Handler
Prefetch_Handler:
    b    Prefetch_Handler
Abort_Handler:
    b    Abort_Handler
FIQ_Handler:
    b    FIQ_Handler

    .global main

    .global IRQ_Handler
    .type IRQ_Handler, "function"
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
IRQ_Handler:
    // IRQ handler code

    .global Reset_Handler
    .type Reset_Handler, "function"
Reset_Handler:
    // Reset_Handler code

    //
    // Branch to application
    // ---------------------
    b    main
```

Create the file `variables.c` containing:

```
int v = 100;
```

Create the file `main.c` containing:

```
#include <stdio.h>

extern int v;

const int ARRAY_SIZE=10;
int x[ARRAY_SIZE];

int main(void) {
    int b;
    int c;
    for (b=0; b<ARRAY_SIZE; b++) {
        c = a * b;
        x[b] = c;
        v++;
        printf("[%d] %d\n", v, x[b]);
    }
}
```

Compile the code:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -c libubsan.c
```

Create the library:

```
armar --create libubsan.a libubsan.o
```

Compile your program and link in the library:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -fsanitize=undefined \
         -fsanitize-minimal-runtime main.c variables.c bootcode.s \
         -Wl,--entry=main -L. -lubsan -o image.axf
```

Related information

-fsanitize, -fno-sanitize on page 84
Overview of Undefined Behavior Sanitizer
Undefined Behavior Sanitizer

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.40 -fsanitize-recover, -fno-sanitize-recover

Controls which checks that are enabled by `-fsanitize` are to be non-fatal. If a check is fatal, then the program halts after the first error of that kind is detected, and `armclang` reports an error.

Default

By default, the non-fatal checks are those enabled by *Undefined Behavior Sanitizer* (UBSan), except for `-fsanitize=return` and `-fsanitize=unreachable`.

Syntax

```
-fsanitize-recover=<option>[,<option>,...]
-fno-sanitize-recover=<option>[,<option>,...]
```

Parameters

`<option>`

`-fsanitize-recover` sets the UBSan check that is enabled by `-fsanitize` to be non-fatal. Specify `-fsanitize-recover=all` to make all checks specified by `-fsanitize` as non-fatal. You can use a comma-separated list to specify multiple checks.


Note

The *Control Flow Integrity* (CFI) option `cfi` is not supported.

Operation

The `-fsanitize-trap` option takes precedence over this option. To override this precedence for a specific check, disable the trapping behavior for that check with `-fno-sanitize-trap`.

Example: Make the bool and vptr UBSan checks non-fatal

```
armclang --target=arm-arm-none-eabi -fsanitize=undefined -fsanitize-
recover=bool,vptr -mcpu=cortex-m3 main.c -o image.axf
```

Related information

-fsanitize, -fno-sanitize on page 84
-fsanitize-trap, -fno-sanitize-trap on page 93
Overview of Control Flow Integrity
Undefined Behavior Sanitizer

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.41 -fsanitize-trap, -fno-sanitize-trap

Enables or disables the traps mode for *Undefined Behavior Sanitizer* (UBSan) checks.

Syntax

```
-fsanitize-trap=<option>[,<option>,...]
-fno-sanitize-trap=<option>[,<option>,...]
```

Parameters

<option>

The UBSan check for which the traps mode is to be enabled. Specify `-fsanitize-trap=all` to use traps mode for all checks requested. You can use a comma-separated list to specify multiple checks.

For a list of the UBSan checks, see Undefined Behavior Sanitizer.

Restrictions

The *Control Flow Integrity* (CFI) option `cfi` is not supported.

Example: Compile with traps

In this mode, when an undefined behavior is detected, a trap instruction is executed. The advantage of this mode is that no runtime library is needed.

Create the file `bootcode.s` containing:

```
    .section StartUp, "ax"
    .balign 0x20
    .global Vectors

Vectors:
    b    Reset_Handler
    b    Undefined_Handler
    b    SVC_Handler
    b    Prefetch_Handler
    b    Abort_Handler
    b .  //Reserved vector
    b    IRQ_Handler
    b    FIQ_Handler
// -------------------------------------------------------------
// Handlers for unused exceptions
// -------------------------------------------------------------

Undefined_Handler:
    b    Undefined_Handler
SVC_Handler:
    b    SVC_Handler
Prefetch_Handler:
    b    Prefetch_Handler
Abort_Handler:
    b    Abort_Handler
FIQ_Handler:
    b    FIQ_Handler

    .global main

    .global IRQ_Handler
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    .type IRQ_Handler, "function"
IRQ_Handler:
    // IRQ handler code

    .global Reset_Handler
    .type Reset_Handler, "function"
Reset_Handler:
    // Reset_Handler code

    //
    // Branch to application
    // ---------------------
    b    main
```

Create the file `main.cpp` containing:

```
int main(int argc, char **argv) {
    int x = 0x7FFFFFFF;
    x += argc;
    return 0;
}
```

Create the linker script `scatter.scat` containing:

```
LR 0x8000
{
    CODE +0
    {
        *(+RO)
    }
}

LR_STACKHEAP 0x20000000
{
    ARM_LIB_STACKHEAP +0 EMPTY 0x2000 {}
}
```

Compile with the `-fsanitize-trap=all -fsanitize=undefined` options:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -fsanitize-trap=all -
fsanitize=undefined \
         main.cpp bootcode.s -Wl,--scatter=scatter.scat -o image_m3.axf
```

The bootcode is used for the implementation of the exception handler and floating point enablement.

Related information

-fsanitize, -fno-sanitize on page 84
-fsanitize-recover, -fno-sanitize-recover on page 93
Overview of Undefined Behavior Sanitizer
Undefined Behavior Sanitizer

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.42 -fshort-enums, -fno-short-enums

Allows the compiler to set the size of an enumeration type to the smallest data type that can hold all enumerator values.

Default

The default is `-fno-short-enums`. That is, the size of an enumeration type is at least 32 bits regardless of the size of the enumerator values.

Syntax

```
-fshort-enums
-fno-short-enums
```

Parameters

None.

Operation

The `-fshort-enums` option can improve memory usage, but might reduce performance because narrow memory accesses can be less efficient than full register-width accesses.


Note

All linked objects, including libraries, must make the same choice. It is not possible to link an object file compiled with `-fshort-enums`, with another object file that is compiled without `-fshort-enums`.


Note

The `-fshort-enums` option is not supported for AArch64. The *Procedure Call Standard for the Arm 64-bit Architecture* states that the size of enumeration types must be at least 32 bits.


Note

ISO C restricts enumerator values to the range of `int`. By default `armclang` does not issue warnings about enumerator values that are too large, but with `-Wpedantic` a warning is displayed.

Example

This example shows the size of four different enumeration types: 8-bit, 16-bit, 32-bit, and 64-bit integers.

```
#include <stdio.h>

// Largest value is 8-bit integer
enum int8Enum  {int8Val1 =0x01, int8Val2 =0x02, int8Val3 =0xF1 };

// Largest value is 16-bit integer
enum int16Enum {int16Val1=0x01, int16Val2=0x02, int16Val3=0xFFF1 };
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
// Largest value is 32-bit integer
enum int32Enum {int32Val1=0x01, int32Val2=0x02, int32Val3=0xFFFFFFF1 };

// Largest value is 64-bit integer
enum int64Enum {int64Val1=0x01, int64Val2=0x02, int64Val3=0xFFFFFFFFFFFFFFF1 };

int main(void)
{
  printf("size of int8Enum  is %zd\n", sizeof (enum int8Enum));
  printf("size of int16Enum is %zd\n", sizeof (enum int16Enum));
  printf("size of int32Enum is %zd\n", sizeof (enum int32Enum));
  printf("size of int64Enum is %zd\n", sizeof (enum int64Enum));
}
```

When compiled without the `-fshort-enums` option, all enumeration types are 32 bits (4 bytes) except for `int64Enum` which requires 64 bits (8 bytes):

```
armclang --target=arm-arm-none-eabi -march=armv8-a enum_test.cpp

size of int8Enum  is 4
size of int16Enum is 4
size of int32Enum is 4
size of int64Enum is 8
```

When compiled with the `-fshort-enums` option, each enumeration type has the smallest size possible to hold the largest enumerator value:

```
armclang -fshort-enums --target=arm-arm-none-eabi -march=armv8-a enum_test.cpp

size of int8Enum  is 1
size of int16Enum is 2
size of int32Enum is 4
size of int64Enum is 8
```

Related information

Procedure Call Standard for the Arm 64-bit Architecture (AArch64)

2.1.43 -fshort-wchar, -fno-short-wchar

`-fshort-wchar` sets the size of `wchar_t` to 2 bytes. `-fno-short-wchar` sets the size of `wchar_t` to 4 bytes.

Default

The default is `-fno-short-wchar`.

Syntax

```
-fshort-wchar
-fno-short-wchar
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

The `-fshort-wchar` option can improve memory usage, but might reduce performance because narrow memory accesses can be less efficient than full register-width accesses.


Note

All linked objects must use the same `wchar_t` size, including libraries. It is not possible to link an object file compiled with `-fshort-wchar`, with another object file that is compiled without `-fshort-wchar`. `armlink` does not report a warning or error when linking with mixed `wchar_t` size.

Example

This example shows the size of the `wchar_t` type:

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  printf("size of wchar_t is %zd\n", sizeof (wchar_t));
  return 0;
}
```

When compiled without the `-fshort-wchar` option, the size of `wchar_t` is 4 bytes:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 wchar_test.c

size of wchar_t is 4
```

When compiled with the `-fshort-wchar` option, the size of `wchar_t` is 2 bytes:

```
armclang -fshort-wchar --target=aarch64-arm-none-eabi -mcpu=cortex-a53 wchar_test.c

size of wchar_t is 2
```

2.1.44 -fsigned-char, -funsigned-char

Set the type of `char` to be signed or unsigned.


Note

This topic describes a [COMMUNITY] feature. See Support level definitions.

Default

The default is `-funsigned-char`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
-fsigned-char
-funsigned-char
```

Parameters

None.

Operation

We recommend the default setting of `-funsigned-char`.

You must take care when mixing translation units that are compiled with and without the [COMMUNITY] `-fsigned-char` 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 supplied with the Arm compilation tools.

Related information

Characters on page 1048

2.1.45 -fsized-deallocation, -fno-sized-deallocation

Enables the C++14 sized deallocation feature.

Default

The default is `-fsized-deallocation` for C++14 onward, `-fno-sized-deallocation` otherwise.

Syntax

```
-fsized-deallocation
-fno-sized-deallocation
```

Parameters

None.

Operation



The C++14 sized deallocation feature is supported with C++11 if the `-fsized-deallocation` command-line option is specified. Sized deallocation is supported by default for C++14 and later.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

When the deallocation feature is enabled the following deallocation functions become available, and are predefined in each translation unit:

```
operator delete(void*, std::size_t)
operator delete(void*, std::size_t, const std::nothrow_t&)
operator delete[](void*, std::size_t)
operator delete[](void*, std::size_t, const std::nothrow_t&)
```

2.1.46 -fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector

Inserts a guard variable onto the stack frame for each vulnerable function or for all functions.

Default

The default is `-fno-stack-protector`.

Syntax

```
-fstack-protector
-fstack-protector-all
-fstack-protector-strong
-fno-stack-protector
```

Parameters

None

Operation

`-fno-stack-protector` disables stack protection.

`-fstack-protector` enables stack protection for vulnerable functions that contain:

- A character array larger than 8 bytes.
- An 8-bit integer array larger than 8 bytes.
- A call to `alloca()` with either a variable size or a constant size bigger than 8 bytes.

`-fstack-protector-all` adds stack protection to all functions regardless of their vulnerability.

`-fstack-protector-strong` enables stack protection for vulnerable functions that contain:

- An array of any size and type.
- A call to `alloca()`.
- A local variable that has its address taken.


Note

If you specify more than one of these options, the last option that is specified takes effect.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The prologue of a function stores a guard variable onto the stack frame. Before returning from the function, the function epilogue checks the guard variable to make sure that it has not been overwritten. A guard variable that is overwritten indicates a buffer overflow, and the checking code alerts the run-time environment.

When a vulnerable function is called with stack protection enabled, 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. For example, a suitable implementation might set this variable to a random value when the program is loaded, and before the first protected function is entered. The value must remain unchanged during the life of the program.

When the checking code detects that the guard variable on the stack has been modified, it notifies the run-time environment by calling the function:

```
void __stack_chk_fail(void);
```

You must provide a suitable implementation for this function. Normally, such a function terminates the program, possibly after reporting a fault.

Optimizations can affect the stack protection. The following are simple examples:

- Inlining can affect whether a function is protected.
- Removal of an unused variable can prevent a function from being protected.

Example: Stack protection

Create the following `main.c` and `get.c` files:

```
// main.c

#include <stdio.h>
#include <stdlib.h>

void *__stack_chk_guard = (void *)0xdeadbeef;

void __stack_chk_fail(void)
{
    fprintf(stderr, "Stack smashing detected.\n");
    exit(1);
}

void get_input(char *data);

int main(void)
{
    char buffer[9];
    get_input(buffer);
    return buffer[0];
}
```

```
// get.c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
#include <string.h>

void get_input(char *data)
{
    strcpy(data, "012345678");
}
```

When `main.c` and `get.c` are compiled with `-fstack-protector`, the array `buffer` is considered vulnerable and stack protection gets applied to the function `main()`. The checking code recognizes the overflow of `buffer` that occurs in `get_input()`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -fstack-protector main.c get.c
```

Running the image displays the following message:

```
Stack smashing detected.
```

Related information

-Rpass on page 213

2.1.47 -fstack-usage

Creates a `.su` file that contains information on the function stack sizes.

Syntax

```
-fstack-usage
```

Parameters

None.

Operation

The `.su` file that is generated has the following syntax:

```
<source_file>:<line_number>:<function_name> <size_in_bytes> <qualifiers>
```

`<source_file>`
The source filename that contains the function definition.

`<line_number>`
The line number of the function definition in the source file.

`<function_name>`
The name of the function.

`<size_in_bytes>`
The size of the fixed size frame object in bytes. This value is not a reliable measure of the total frame size.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<qualifiers>

One of:

static

The function statically manipulates the stack. Because the frame size of the function is static, then the size information is an accurate reflection of the frame size.

dynamic

The function dynamically manipulates the stack. Therefore, the frame size of the function can only be determined at run-time. For example, because objects have a variable size.

Example: Get the stack usage for a variable length array

Create the file `main.c` containing:

```
#include "stdio.h"

void my_array(int a, int size)
{
    int b, c;
    int array[size];
    for (b=0; b<size; b++) {
        c = a * b;
        array[b] = c;
        printf("%d\n", array[b]);
    }
}

int main(void) {
    my_array(5, 10);
    return 0;
}
```

Compile with the following command:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -fstack-usage -c main.c -o
 image.o
```

The generated `image.su` file contains:

```
main.c:3:my_array 48 dynamic
main.c:14:main 16 static
```

Related information

-fsanitize-minimal-runtime on page 89

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.48 -fstrict-aliasing, -fno-strict-aliasing

Instructs the compiler to apply the strictest aliasing rules available.

Default

`-fstrict-aliasing` is implicitly enabled at `-O1` or higher. It is disabled at `-O0`, or when no optimization level is specified.

Syntax

```
-fstrict-aliasing
-fno-strict-aliasing
```

Parameters

None.

Operation

When optimizing at `-O1` or higher, this option can be disabled with `-fno-strict-aliasing`.


Note

Specifying `-fstrict-aliasing` on the command line has no effect, because it is either implicitly enabled, or automatically disabled, depending on the optimization level that is used.

Examples

In the following example, `-fstrict-aliasing` is enabled:

```
armclang --target=aarch64-arm-none-eabi -O2 -c hello.c
```

In the following example, `-fstrict-aliasing` is disabled:

```
armclang --target=aarch64-arm-none-eabi -O2 -fno-strict-aliasing -c hello.c
```

In the following example, `-fstrict-aliasing` is disabled:

```
armclang --target=aarch64-arm-none-eabi -c hello.c
```

2.1.49 -fsysv, -fno-sysv

Enables or disables the generation of code suitable for the SysV linking model.

Default

The default is `-fno-sysv`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
-fsysv
-fno-sysv
```

Parameters

None.

Restrictions

Arm® Compiler for Embedded does not support dynamic linking when targeting Cortex®-M processors.

Operation

`-fsysv` causes the compiler to disable bare-metal optimizations that are not suitable for the SysV linking model.

`-fsysv` causes the compiler to invoke `armlink` with the `--sysv` option when performing the link step.

Related information

SysV Dynamic Linking

2.1.50 -ftls-model

Specifies the *Thread Local Storage* (TLS) model to use.



Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.

Default

The default value for `-ftls-model` depends on the value of `-fpic`:

- The default without `-fpic` is `initial-exec`.
- The default with `-fpic` is `global-dynamic`.

Syntax

```
-ftls-model=<model>
```

Parameters

`<model>`

One of the following, ordered from most restrictive to least restrictive:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

local-exec

Handles access to symbols that exist within the executable or module at static link time, where all offsets are resolved in relation to the thread pointer. It defines a single storage area for all TLS variables.

initial-exec

Handles access to symbols from external modules that are initially loaded, where all offsets are run-time constants.

global-dynamic

The most generic model, which handles external access to dynamically loaded modules. This model is sometimes referred to as the general dynamic model in the TLS specification and ABI documents.


Note

A fourth model, `local-dynamic`, is not supported. If you use `-ftls-model=local-dynamic`, then access is handled in the same way as `-ftls-model=global-dynamic`.


Tip

We recommend using `-ftls-model=local-exec` for static linking, or if you know that all thread local variables are defined in the executable.

Operation

The compiler can generate code sequences to reference the TLS variables using different models depending on where the TLS variable is defined and loaded. Each model is a trade-off between flexibility and efficiency. The compiler might use a more restrictive model when it can determine it is safe to do so.

The `__thread` keyword and `thread_local` keyword are equivalent. The `__thread` keyword is universally accepted as a GNU extension in both C and C++. The `thread_local` keyword is supported for C++ with `-std=c++11` or later. The `thread_local` keyword is only supported for C in Arm® Compiler for Embedded version 6.19 and later when used with the `-std=c2x` [COMMUNITY] feature for C23 support.

The compiler attempts to select the appropriate TLS model for each `__thread` variable as follows:

- If `-fpic` is used, then the compiler makes the assumption that the code being compiled can be used in a shared object. Therefore, the `general-dynamic` model is used.
- If `-fpic` is not used, then the compiler makes the assumption that the code being compiled is for a statically linked executable. Either `local-exec` or `initial-exec` is used.
  - `local-exec` is used for `__thread` variables that are defined and used within the same source file.
  - `initial-exec` is used for `__thread` variables that are defined using `extern`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

That is, the compiler selects the least flexible and most efficient TLS model that is guaranteed to work in each case.

You can override these choices using either the `-ftls-model` command-line option or `tls_model` variable attribute if you know it is safe to do so.

Example: AArch64 TLS local-exec static linking example

For an example of AArch64 TLS `local-exec` static linking with source code that you can build and run, see AArch64 TLS local-exec static linking example.

Related information

-fpic, -fno-pic on page 79
-mtls-size on page 194
-mtp on page 195
__attribute__((tls_model("model"))) variable attribute on page 298
Thread Local Storage

2.1.51 -ftrapv

Instructs the compiler to generate traps for signed arithmetic overflow on addition, subtraction, and multiplication operations.

Default

`-ftrapv` is disabled by default.

Syntax

```
-ftrapv
```

Parameters

None.

Operation

The compiler inserts code that checks for overflow and traps the overflow with an undefined instruction. An undefined instruction handler must be provided for the overflow to get caught at run-time.


Note

When both `-fwrapv` and `-ftrapv` are used in a single command, the furthest-right option overrides the other.

For example, here `-ftrapv` overrides `-fwrapv`:

```
armclang --target=aarch64-arm-none-eabi -fwrapv -c -ftrapv hello.c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.52 -ftrivial-auto-var-init

Initializes trivial automatic variables to a fixed repeating pattern or to zero.

Default

The default is `-ftrivial-auto-var-init=uninitialized`.

Syntax

```
-ftrivial-auto-var-init=<option>
```

Parameters

`<option>`

One of:

`pattern`

Initializes trivial automatic variables with a repeated pattern. The pattern depends on the type of the variable and the state:

Table 2-11: Patterns for variables in AArch32 and AArch64 states

Variable type	AArch32 state pattern	AArch64 state pattern
`int`	`0xFFFFFFFF`	`0xAAAAAAAA`
Pointer	`0xFFFFFFFF`	`0xAAAAAAAAAAAAAAAA`
`char`	`0xFF`	`0xAA`
`short`	`0xFFFF`	`0xAAAA`
`float`	`0xFFFFFFFF`	`0xFFFFFFFF`
`double`	`0xFFFFFFFFFFFFFFFF`	`0xFFFFFFFFFFFFFFFF`
`struct`	Each individual member is initialized using the relevant pattern	Each individual member is initialized using the relevant pattern
C++ zero-length `struct`	`0xFF`	`0xAA`
`union`	Uses the same pattern as the largest member	Uses the same pattern as the largest member
Array	Uses the same pattern as the underlying type	Uses the same pattern as the underlying type

`uninitialized`

All trivial automatic variables are uninitialized.

`zero`

Initializes trivial automatic variables with 0.

Use the `__attribute__((uninitialized))` variable attribute to ensure individual automatic variables are not initialized when using the `pattern` or `zero` options.

Related information

__attribute__((uninitialized)) variable attribute on page 300

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.53 -fvectorize, -fno-vectorize

Enables or disables the generation of Advanced SIMD (Neon®), *M-profile Vector Extension* (MVE), and *Scalable Vector Extension* (SVE) vector instructions directly from C or C++ code at optimization levels `-O1` and higher.

Default

The default depends on the optimization level in use.

At optimization level `-O0` (the default optimization level), `armclang` never performs automatic vectorization. The `-fvectorize` and `-fno-vectorize` options are ignored.

At optimization level `-O1`, the default is `-fno-vectorize`. Use `-fvectorize` to enable automatic vectorization. When using `-fvectorize` with `-O1`, vectorization might be inhibited in the absence of other optimizations which might be present at `-O2` or higher.

At optimization level `-O2` and above, the default is `-fvectorize`. Use `-fno-vectorize` to disable automatic vectorization.

Using `-fno-vectorize` does not necessarily prevent the compiler from emitting Advanced SIMD, MVE, or SVE instructions. The compiler or linker might still introduce Advanced SIMD, MVE, or SVE instructions, such as when linking libraries that contain these instructions.

Syntax

```
-fvectorize
-fno-vectorize
```

Parameters

None.

Examples

This example enables automatic vectorization with optimization level `-O1`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -fvectorize -O1 -c file.c
```

To prevent the compiler from emitting Advanced SIMD instructions for AArch64 targets, specify `+nosimd` using `-march` or `-mcpu`. For example:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a+nosimd -O2 file.c -c -S -o
 file.s
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

To prevent the compiler from emitting Advanced SIMD instructions for AArch32 targets, set the option `-mfpu` to the correct value that does not include Advanced SIMD, for example `fp-armv8`:

```
armclang --target=aarch32-arm-none-eabi -march=armv8-a -mfpu=fp-armv8 -O2 file.c -c
 -S -o file.s
```

Related information

-c (armclang) on page 43
-O (armclang) on page 203

2.1.54 -fvisibility

Sets the default visibility of ELF symbol definitions to the specified option. This option does not affect the visibility of reference, `extern`, ELF symbols.

Default

The default type is `-fvisibility=hidden`.

Syntax

```
-fvisibility=<visibility_type>
```

Parameters

`<visibility_type>`

One of the following:

`default`

All global symbols that are not declared with the `hidden` or `protected` visibility attribute are public and corresponds to external linkage. External or local symbols are not affected.

`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.


Note

`extern` declarations are visible, and all other symbols are hidden.

`protected`

The symbol is placed into the dynamic symbol table, but references within the defining module bind to the local symbol. That is, another module cannot override the symbol.

The visibility impacts partial and shared library output.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

You can override `-fvisibility` in code with the `__attribute__((visibility("<visibility_type>")))` attribute.

Related information

__attribute__((visibility("visibility_type"))) function attribute on page 285
__attribute__((visibility("visibility_type"))) variable attribute on page 302

2.1.55 -fwrapv

Instructs the compiler to assume that signed arithmetic overflow of addition, subtraction, and multiplication, wraps using two's-complement representation.

Default

`-fwrapv` is disabled by default.

Syntax

```
-fwrapv
```

Parameters

None.

Operation

When both `-fwrapv` and `-ftrapv` are used in a single command, the furthest-right option overrides the other.

For example, here `-fwrapv` overrides `-ftrapv`:

```
armclang --target=aarch64-arm-none-eabi -ftrapv -c -fwrapv hello.c
```

2.1.56 -g, -g<level>, -gdwarf-<version> (armclang)

Controls the generation of debug tables for source-level debugging.

Default

By default, `armclang` does not produce debug information. When you use `-g`, the default `<level>` is 2 and the default `<version>` is 4.

Syntax

```
-g
-g<level>
-gdwarf-<version>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<level>

The amount of detail to include in the debug information. A higher value for <level> generates more detailed debug information. Valid values are:

0

Disables the generation of debug information.

1

Generates line table debug information only.

2

Generates debug information.

3

Generates debug information.

<version>

The DWARF format to produce. Valid values are 2, 3, 4, and 5.

Operation

When compiling C and C++ code, `armclang` generates `.cfi` directives to provide debug information that is compatible with the specified DWARF standard. However, the `armclang` integrated assembler does not generate `.cfi` directives. Therefore, you must add the `.cfi` directives to your GNU-syntax assembly source. See How to get a backtrace through assembler functions for more information.

Use a compatible debugger to load, run, and debug images. For example, Arm® Debugger is compatible with DWARF 4. Compile with the `-g` or `-gdwarf-4` options to debug with Arm Debugger.


Note

The Keil® µVision® debugger is compatible with DWARF 4.

Legacy and third-party tools might not support DWARF 4 debug information. In this case you can specify the level of DWARF conformance required using the `-gdwarf-2` or `-gdwarf-3` options.

Because the DWARF 4 specification supports language features that are not available in earlier versions of DWARF, the `-gdwarf-2` and `-gdwarf-3` options must only be used for backwards compatibility.


Note

DWARF standard versions not documented here are not supported. Community options might not work as expected because of dependencies on other tools in the toolchain.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`-gdwarf-<version>` overrides `-g<level>` and generates debug tables at the default `-g` or `-g2` level.

Example: Specifying -gdwarf-3 and -gdwarf-2

```
armclang --target=arm-arm-none-eabi -march=armv8-a -gdwarf-3 -gdwarf-2
```

Produces DWARF 2 debug, because `-gdwarf-2` overrides `-gdwarf-3`. Generates debug tables at the default `-g` or `-g2` level.

Example: Specifying -g before -gdwarf-2

```
armclang --target=arm-arm-none-eabi -march=armv8-a -g -gdwarf-2
```

Produces DWARF 2 debug, because `-gdwarf-2` overrides the default DWARF 4 implied by `-g`. Generates debug tabes at the default `-g` or `-g2` level.

Example: Specifying -g1 before -gdwarf-2

```
armclang --target=arm-arm-none-eabi -march=armv8-a -g1 -gdwarf-2
```

Produces DWARF 2 debug, because `-gdwarf-2` overrides the default DWARF 4 implied by `-g1`. Generates debug tables at the default `-g` or `-g2` level, because `-gdwarf-<version>` also overrides the `<level>` specified in `-g<level>`.

Example: Specifying -gdwarf-2 before -g1

```
armclang --target=arm-arm-none-eabi -march=armv8-a -gdwarf-2 -g1
```

Produces DWARF 2 debug. Generates debug tables at `-g1` level.

Example: Specifying -g0 after -gdwarf-2

```
armclang --target=arm-arm-none-eabi -march=armv8-a -gdwarf-2 -g0
```

Produces no debug tables.

2.1.57 -I

Adds the specified directory to the list of places that are searched to find include files.

Syntax

```
-I<dir>
-I <dir>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<dir>

A directory to search for included files.

Operation

Use multiple `-I` options to specify multiple search directories.

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

2.1.58 -include

Includes the source code of the specified file at the beginning of the compilation.

Syntax

```
-include <filename>
```

Parameters

<filename>

The name of the file whose source code is to be included.



Note

Any `-D`, `-I`, and `-U` options on the command line are always processed before `-include <filename>`.

Related information

-D (armclang) on page 44
-I on page 113
-U on page 222

2.1.59 -isystem <directory>

The `-isystem` command-line option allows you to specify a directory as a system header directory. This option means that headers included with `-isystem` are treated as system headers. This option is useful when you want to include specific system headers from non-standard locations.

Syntax

```
-isystem <directory>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<directory>

The directory that you want to add to the system headers.

Operation

Searches in the specified directory for both the system paths, for example `#include <header.h>`, and the regular paths, for example `#include "header.h"`.

`-isystem` is useful when using `-nobuiltininc`, which prevents the inclusion of the default system headers.

Related information

-nobuiltininc on page 198

2.1.60 -L

Specifies a list of paths that the linker searches for user libraries.

Syntax

```
-L<dir> [,<dir>,...]
```

Parameters

<dir>[,<dir>,...]

A comma-separated list of directories to be searched for user libraries.

At least one directory must be specified.

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

If you use `armclang` to invoke the linker, `armclang` automatically translates this option to `--userlibpath` and passes it to `armlink`. If you specify the `armclang` option `-c`, you must invoke `armlink` separately with the `--userlibpath` option.

See --userlibpath=pathlist for information.



Note

The `-L` option has no effect when used with the `-c` option, that is when not linking.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.61 -l

Add the specified library to the list of searched libraries.

Syntax

```
-l<name>
```

Parameters

<name>

The name of the library, for example `lib<name>.a`.

If you use `armclang` to invoke the linker, `armclang` automatically translates this option to `--library` and passes it to `armlink`. If you specify the `armclang` option `-c`, you must invoke `armlink` separately with the `--library` option.

See the *Arm Compiler for Embedded Reference Guide* for information about the `armlink` option `--library`.


Note

The `-l` option has no effect when used with the `-c` option, that is when not linking.

2.1.62 -M, -MM

Produces a list of makefile dependency rules suitable for use by a make utility.

Syntax

```
-M
-MM
```

Parameters

None.

Operation

`armclang` executes only the preprocessor step of the compilation or assembly. By default, output is on the standard output stream.

If you specify multiple source files, a single dependency file is created.

`-M` lists both system header files and user header files.

`-MM` lists only user header files.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

The `-MT` option lets you override the target name in the dependency rules.

Note

To compile or assemble the source files and produce makefile dependency rules, use the `-MD` or `-MMD` option instead of the `-M` or `-MM` option respectively.

Note

Only dependencies visible to the preprocessor are included. Files added using the GNU assembler syntax `.incbin` or `.include` directives (or `armasm` syntax `INCBIN`, `INCLUDE`, or `GET` directives) are not included.

Example

You can redirect output to a file using standard UNIX and Windows notation, the `-o` option, or the `-MF` option. For example:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -M source.c > deps.mk
armclang --target=arm-arm-none-eabi -march=armv8-a -M source.c -o deps.mk
armclang --target=arm-arm-none-eabi -march=armv8-a -M source.c -MF deps.mk
```

Related information

-MD, -MMD on page 117
-o (armclang) on page 206

2.1.63 -MD, -MMD

Compiles or assembles source files and produces a list of makefile dependency rules suitable for use by a make utility.

Syntax

```
-MD
-MMD
```

Parameters

None.

Operation

`armclang` creates a makefile dependency file for each source file, using a `.d` suffix. Unlike `-M` and `-MM`, that cause compilation or assembly to stop after the preprocessing stage, `-MD` and `-MMD` allow for compilation or assembly to continue.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`-MD` lists both system header files and user header files.

`-MMD` lists only user header files.


Note

Only dependencies visible to the preprocessor are included. Files added using the GNU assembler syntax `.incbin` or `.include` directives (or `armasm` syntax `INCBIN`, `INCLUDE`, or `GET` directives) are not included.

Example

The following example creates makefile dependency lists `test1.d` and `test2.d` and compiles the source files to an image with the default name, `a.out`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -MD test1.c test2.c
```

Related information

-M, -MM on page 116
-MF on page 118
-MT on page 121

2.1.64 -MF

Specifies a filename for the makefile dependency rules produced by the `-M` and `-MD` options.

Syntax

```
-MF  <filename>
```

Parameters

`<filename>`

Specifies the filename for the makefile dependency rules.


Note

The `-MF` option only has an effect when used in conjunction with one of the `-M`, `-MM`, `-MD`, or `-MMD` options.

Operation

The `-MF` option overrides the default behavior of sending dependency generation output to the standard output stream, and sends output to the specified filename instead.

`armclang -MD` sends output to a file with the same name as the source file by default, but with a `.d` suffix. The `-MF` option sends output to the specified filename instead. Only use a single source file with `armclang -MD -MF`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Send makefile dependency rules to standard output, without compiling the source

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M source.c
```

Example: Save makefile dependency rules to deps.mk, without compiling the source

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M source.c -MF deps.mk
```

Example: Compile the source and saves makefile dependency rules to source.d (using the default file naming rules)

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -MD source.c
```

Example: Compile the source and saves makefile dependency rules to deps.mk

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -MD source.c -MF deps.mk
```

Related information

-M, -MM on page 116
-MD, -MMD on page 117
-MT on page 121

2.1.65 -MG

Prints dependency lines for header files even if the header files are missing.

Syntax

```
-MG
```

Parameters

None.

Restrictions

The `-MG` option only has an effect when used with one of the following options: `-M` or `-MM`.

Operation

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

Example

`source.c` contains a reference to a missing header file `header.h`:

```
#include <stdio.h>
#include "header.h"

int main(void){
    puts("Hello world\n");
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    return 0;
}
```

This first example is compiled without the `-MG` option, and results in an error:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M source.c

source.c:2:10: fatal error: 'header.h' file not found
#include "header.h"
         ^
1 error generated.
```

This second example is compiled with the `-MG` option, and the error is suppressed:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M -MG source.c

source.o: source.c \
  /include/stdio.h \
  header.h
```

Related information

-M, -MM on page 116

2.1.66 -MP

Emits placeholder dependency rules.

Syntax

```
-MP
```

Parameters

None.

Restrictions

The `-MP` option only has an effect when used in conjunction with the `-M`, `-MD`, `-MM`, or `-MMD` options.

Operation

The placeholder dependency rules work around `make` errors that are generated if you remove header files without a corresponding update to the makefile.

Example: Send dependency rules to standard output, without compiling the source

Create the file `source.c` that includes a header file:

```
#include <stdio.h>

int main(void){
    puts("Hello world\n");
    return 0;
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Compiling without the -MP option results in a dependency rule for source.o

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M source.c

source.o: source.c \
  /include/stdio.h
```

Example: Compiling with the -MP option results in a dependency rule for source.o and a dummy rule for the header file

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M -MP source.c

source.o: source.c \
  /include/stdio.h

/include/stdio.h:
```

Related information

-M, -MM on page 116
-MD, -MMD on page 117

2.1.67 -MT

Changes the target of the makefile dependency rule produced by dependency generating options.

Syntax

```
-MT
```

Parameters

None.

Restrictions

The `-MT` option only has an effect when used in conjunction with either the `-M`, `-MM`, `-MD`, or `-MMD` options.

Operation

By default, `armclang -M` creates makefile dependencies rules based on the source filename:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M test.c

test.o: test.c header.h
```

The `-MT` option renames the target of the makefile dependency rule:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M test.c -MT foo

foo: test.c header.h
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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, the `-MT` option renames the target of all dependency rules:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M test1.c test2.c -MT foo

foo: test1.c header.h
foo: test2.c header.h
```

Specifying multiple `-MT` options creates multiple targets for each rule:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M test1.c test2.c -MT foo
 -MT bar

foo bar: test1.c header.h
foo bar: test2.c header.h
```

Related information

-M, -MM on page 116
-MD, -MMD on page 117
-MF on page 118

2.1.68 -mabi=<name>

Specifies the calling convention for AArch64 state.

Default

The default is `aapcs`.

Syntax

```
-mabi=<name>
```

Parameters

`<name>`

One of the following:

`aapcs`

Use the default AArch64 *Application Binary Interface* (ABI), which passes some function arguments in floating-point registers. If the target does not have floating-point registers, then the compiler reports an error if the source code contains any floating-point types.

`aapcs-soft`

Use the soft-float ABI variant, which does not use any floating-point registers. If the target has floating-point registers and you use this option, then `armclang` outputs an error.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Restrictions

Valid only for AArch64 state with the following exceptions:

- `-mabi=aapcs-soft` is not supported for A-profile targets in AArch64 state. The `aapcs-soft` ABI is defined only for Arm®v8-R AArch64 targets. For more information, see the *Soft-float* section of the Procedure Call Standard for the Arm 64-bit Architecture.
- `-mabi=aapcs-soft` is not supported for C++ source language modes.

2.1.69 -march

Targets an Arm® architecture profile, generating generic code that runs on any processor of that architecture.


Note

This topic includes descriptions of [ALPHA] and [BETA] features. See Support level definitions.


Note

Avoid specifying both the architecture (`-march`) and the processor (`-mcpu`) because specifying both has the potential to cause a conflict. The compiler infers the correct architecture from the processor.

Default

When compiling with `--target=aarch64-arm-none-eabi`, the default is `-march=armv8-a`.

When compiling with `--target=arm-arm-none-eabi`, the default is unsupported. Ensure that you always use one of the `-march` or `-mcpu` options when compiling with `--target=arm-arm-none-eabi`.


Note

When compiling with `--target=arm-arm-none-eabi` and without `-march` or `-mcpu`, the compiler reports:

```
warning: 'armv4t' is unsupported in this version of the product
```

Syntax

To specify a target architecture:

```
-march=<name>
```

To enable or disable optional extensions supported by an architecture:

```
-march=<name>[+[no]<feature>+...]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

To view a list of all the supported architectures:

```
-march=list
```

Parameters

<name>

One of the following architecture names:

Architecture <name>	Valid targets	Description
armv9-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv9-A application architecture profile.
armv9.1-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv9.1-A application architecture profile.
armv9.2-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv9.2-A application architecture profile.
armv9.3-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv9.3-A application architecture profile.
armv9.4-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv9.4-A application architecture profile.
armv9.5-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv9.5-A application architecture profile.
armv9.6-a	--target=aarch64-arm-none-eabi	Armv9.6-A application architecture profile. The 64-bit Armv9.6-A architecture support level is [BETA].
armv8-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv8-A application architecture profile.
armv8.1-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv8.1-A application architecture profile.
armv8.2-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv8.2-A application architecture profile.
armv8.3-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv8.3-A application architecture profile.
armv8.4-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv8.4-A application architecture profile.
armv8.5-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv8.5-A application architecture profile.
armv8.6-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv8.6-A application architecture profile.
armv8.7-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv8.7-A application architecture profile.
armv8.8-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv8.8-A application architecture profile.
armv8.9-a	--target=aarch64-arm-none-eabi and --target=arm-arm-none-eabi	Armv8.9-A application architecture profile.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Architecture <name>	Valid targets	Description
armv8-r	--target=aarch64-arm-none-eabi	Armv8-R AArch64 real-time architecture profile. Armv8-R AArch64 state with hardware floating-point. Armv8-R AArch64 implementations without hardware floating-point are not supported.
armv8-r	--target=arm-arm-none-eabi	Armv8-R AArch32 real-time architecture profile.
armv8-m.base	--target=arm-arm-none-eabi	Armv8-M microcontroller architecture profile without the Main Extension. Derived from the Armv6-M architecture.
armv8-m.main	--target=arm-arm-none-eabi	Armv8-M microcontroller architecture profile with the Main Extension. Derived from the Armv7-M architecture.
armv8.1-m.main	--target=arm-arm-none-eabi	Armv8.1 microcontroller architecture profile with the Main Extension.
armv7-a	--target=arm-arm-none-eabi	Armv7-A application architecture profile.
armv7-r	--target=arm-arm-none-eabi	Armv7-R real-time architecture profile.
armv7-m	--target=arm-arm-none-eabi	Armv7-M microcontroller architecture profile.
armv7e-m	--target=arm-arm-none-eabi	Armv7-M microcontroller architecture profile with DSP extension.
armv6-m	--target=arm-arm-none-eabi	Armv6-M microcontroller architecture profile.

<feature>

An optional architecture feature that might be enabled or disabled by default depending on the architecture or processor.

Caution

The compiler and integrated assembler do not prevent the use of +<feature> modifiers with an incompatible target. armclang is not guaranteed to report an error or warning for invalid architectural feature combinations. Therefore, you must manually ensure that you only specify valid +<feature> and +no<feature> modifiers for the -march and -mcpu options. See the example section *Use of an invalid feature with the Armv8-A architecture* for an example of using an invalid feature combination.

Note

- If a feature is mandatory in an architecture, then that feature is enabled by default. For AArch32 state inputs only, you can use fromelf --decode_build_attributes to determine whether the feature is enabled.
- For some Arm processors, -mcpu supports specific combinations of the architecture features. See Supported architecture feature combinations for specific processors for more information.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`+<feature>` enables the feature if it is disabled by default. `+<feature>` has no effect if the feature is already enabled by default.

`+no<feature>` disables the feature if it is enabled by default. `+no<feature>` has no effect if the feature is already disabled by default.

Use `+<feature>` or `+no<feature>` to explicitly enable or disable an optional architecture feature.

You can specify one or more features that an architecture supports.

Table 2-13: AArch64 state A-profile features

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`aes`	FEAT_AES FEAT_PMULL	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`b16b16`	FEAT_SVE_B16B16	Armv9-A	-	SVE2.1 and SME2.1 non-widening BFloat16 instructions. See Scalable Matrix Extension for more information.	[ALPHA]
`bf16`	FEAT_BF16	Armv8.2-A	Armv8.6-A	BFloat16 Floating-point extension. See Floating-point extensions for more information.	Supported
`brbe`	FEAT_BRBE	Armv9.2-A	-	Invalidate the Branch Record Buffer extension. Enables support for the `BRB IALL` and `BRB INJ` instructions and system registers.	Supported
`cmpbr`	FEAT_CMPBR	Armv9.5-A	Armv9.6-A	Compare and Branch instructions.	[BETA]
`cpa`	FEAT_CPA	Armv9.4-A	Armv9.5-A	*Checked Pointer Arithmetic* (CPA) extension.	Supported
`crc`	FEAT_CRC32	Armv8-A	Armv8.1-A	*Cyclic Redundancy Check* (CRC) extension.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`crypto`	FEAT_AES FEAT_PMULL FEAT_SHA1 FEAT_SHA256 Armv8.4-A and later also includes: FEAT_SHA512 FEAT_SHA3 FEAT_SM3 FEAT_SM4	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`cssc`	FEAT_CSSC	Armv8.7-A	Armv8.9-A Armv9.4-A	*Common Short Sequence Compression* (CSSC) instructions.	Supported
`d128`	FEAT_D128	Armv9.4-A	-	128-bit page table descriptor. `d128` enables FEAT_D128 but also FEAT_SYSREG128 and FEAT_SYSINSTR128. `d128` also enables the `lse128` feature, FEAT_LSE128.	Supported
`dotprod`	FEAT_DotProd	Armv8.2-A	Armv8.4-A	Advanced *Single Instruction Multiple Data* (SIMD) dot product instructions. Enables the `SDOT` and `UDOT` instructions.	Supported
`f32mm`	FEAT_F32MM	Armv8.2-A	-	Matrix Multiply extension. See Matrix Multiplication extension for more information.	Supported
`f64mm`	FEAT_F64MM	Armv8.2-A	-	Matrix Multiply extension. See Matrix Multiplication extension for more information.	Supported
`f8f16mm`	FEAT_F8F16MM	Armv9.2-A	-	FP8 to half-precision Matrix Multiplication. `f8f16mm` enables `f8f32mm` and `fp8dot2`. See Matrix Multiplication extension for more information.	[BETA]

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`f8f32mm`	FEAT_F8F32MM	Armv9.2-A	-	FP8 to single-precision Matrix Multiplication. `f8f32mm` enables `fp8dot4`. See Matrix Multiplication extension for more information.	[BETA]
`faminmax`	FEAT_FAMINMAX	Armv9.5-A	Armv9.5-A	Floating-point absolute minimum and maximum. Advanced SIMD, SVE2, and SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`flagm`	FEAT_FlagM	Armv8.2-A	Armv8.4-A	Flag Manipulation instructions.	Supported
`fp`	FEAT_FP	Armv8-A	-	Floating-point extension. See Floating-point extensions for more information.	Supported
`fp8`	FEAT_FP8	Armv9.2-A	-	Advanced SIMD, SVE2, and SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`fp8dot2`	FEAT_FP8DOT2	Armv9.2-A	-	Advanced SIMD, SVE2, and SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`fp8dot4`	FEAT_FP8DOT4	Armv9.2-A	-	Advanced SIMD, SVE2, and SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`fp8fma`	FEAT_FP8FMA	Armv9.2-A	-	Advanced SIMD, SVE2, and SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`fp16`	FEAT_FP16	Armv8.2-A	-	Half-precision Floating-point extension. See Floating-point extensions for more information.	Supported
`fp16fml`	FEAT_FHM	Armv8.2-A	-	Half-precision floating-point multiply with add or multiply with subtract extension. See Floating-point extensions for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`fprcvt`	FEAT_FPRCVT	Armv9.5-A	Armv9.6-A	Floating-point to/from integer in scalar FP register.	[BETA]
`gcs`	FEAT_GCS	Armv9.4-A	-	The *Guarded Control Stack* (GCS) extension enables you to securely store return addresses, and provides a method to check that a return address is not compromised.	[ALPHA]
`hbc`	FEAT_HBC	-	Armv8.8-A Armv9.3-A	Hinted Conditional Branches extension. This extension is disabled by default for the Armv8.7-A and Armv9.2-A architectures.	Supported
`i8mm`	FEAT_I8MM	Armv8.2-A	Armv8.6-A Armv9.1-A	Matrix Multiply extension. See Matrix Multiplication extension for more information.	Supported
`ite`	FEAT_ITE	Armv9.4-A	-	Instrumentation Extension. This extension enables a `TRCIT xN` instruction for software to add data into the ETE trace stream.	Supported
`ls64`	FEAT_LS64 FEAT_LS64_V FEAT_LS64_ACCDATA	Armv8.7-A	-	Accelerator support extension for 64-byte atomic loads and stores.	Supported
`lse128`	FEAT_LSE128	Armv9.4-A	-	128-bit atomic instructions. Provides instructions that can atomically read-modify-write 128 bits of a memory location. 128-bit read-modify-write of a memory location is required for editing the 128-bit translation table descriptor.	Supported
`lsfe`	FEAT_LSFE	Armv9.5-A	-	Large System Float Extension.	[BETA]
`lsui`	FEAT_LSUI	Armv9.5-A	Armv9.6-A	Unprivileged load store instructions.	[BETA]

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`lut`	FEAT_LUT	Armv9.2-A	Armv9.5-A	Lookup-table instructions for Advanced SIMD and optionally SVE2 and SME2 if those extensions are implemented.	Supported
`memtag`	FEAT_MTE FEAT_MTE2	Armv8.5-A	-	Memory tagging extension. See Overview of memory tagging for more information.	Supported
`mops`	FEAT_MOPS	-	Armv8.8-A Armv9.3-A	Memory Operations extension. Adds support for instructions to standardize the `memcpy`, `memmove`, and `memset` operations. The extension is disabled by default for the Armv8.7-A and Armv9.2-A architectures. See A-profile Memory Operations Extension for more information.	Supported
`occmo`	FEAT_OCCMO	Armv9.5-A	Armv9.6-A	Outer Cacheable Cache Maintenance Operation.	[BETA]
`pauth`	FEAT_PAuth	Armv8.2-A	Armv8.3-A	Pointer authentication extension. You can set this feature when using -mbranch-protection to ensure pointer authentication works in the same way on Armv8.2-A architectures as it does on Armv8.3-A and later architectures.	Supported
`pauth-lr`	FEAT_PAuth_LR	Armv9.5-A	-	Pointer authentication instructions that enable signing of the *Link Register* (LR) using the *Stack Pointer* (SP) and the *Program Counter* (PC) as diversifiers. See -mbranch-protection for more information.	Supported
`pcdphint`	FEAT_PCDPHINT	Armv9.5-A	-	Producer-Consumer Data Placement Hints	[BETA]

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`pmuv3`	FEAT_PMUv3	Armv8-A	-	*Performance Monitor Extension v3* (PMUv3). This option is enabled by default if you use `-mcpu` to enable code generation for a specific CPU that implements PMUv3. However, if you use `-march` to target an architecture profile, then `pmuv3` is not enabled by default and must be specified explicitly when required. This option only relates to the code generated by the compiler. The assembler always accepts PMUv3 system registers.	Supported
`pops`	FEAT_PoPS	Armv9.5-A	-	Point of Physical Storage	[BETA]
`predres`	FEAT_SPECRES	Armv8-A	Armv8.5-A	Speculation restriction instructions. Enable instructions to prevent data prediction. See Prevention of Speculative execution and data prediction for more information.	Supported
`predres2`	FEAT_SPECRES2	Armv8-A	Armv8.9-A Armv9.4-A	Enhanced speculation restriction instructions. See Prevention of Speculative execution and data prediction for more information.	Supported
`profile`	FEAT_SPE	Armv8.2-A	-	Statistical Profiling Extension.	Supported
`ras`	FEAT_RAS FEAT_RASv1p1	Armv8-A	Armv8.2-A	Reliability, Availability, and Serviceability extension.	Supported
`rasv2`	FEAT_RASv2	-	Armv8.9-A Armv9.4-A	Reliability, Availability, and Serviceability extension version 2. `rasv2` adds an additional system register to `ras`.	Supported
`rcpc`	FEAT_LRCPC	-	Armv8.3-A	Release Consistent Processor Consistent extension.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`rcpc3`	FEAT_LRCPC3	Armv8.2-A	-	Additional support for the Release Consistent Processor Consistent extension. `rcpc3` also enables the `rcpc` feature.	Supported
`rme-gpc3`	FEAT_RME_GPC3	Armv9.5-A	-	RME Granule Protection Check 3 extension. `rme-gpc3` enables `rme-gpc` and `rme-gpc2`.	[BETA]
`rng`	FEAT_RNG	Armv8.5-A	-	Random Number Generation extension.	Supported
`sb`	FEAT_SB	Armv8-A	Armv8.5-A	Speculation barrier instruction. See Prevention of Speculative execution and data prediction for more information.	Supported
`sha2`	FEAT_SHA256 FEAT_SHA1	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`sha3`	FEAT_SHA3 FEAT_SHA512	Armv8.2-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`simd`	FEAT_AdvSIMD	Armv8-A	-	Advanced SIMD extension.	Supported
`sm4`	FEAT_SM4	Armv8.2-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`sme`	FEAT_SME	Armv9.2-A	-	*Scalable Matrix Extension* (SME). See Scalable Matrix Extension for more information.	Supported
`sme2p1`	FEAT_SME2p1	Armv9.2-A	-	SME version 2.1. See Scalable Matrix Extension for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`sme2p2`	FEAT_SME2p2	Armv9.5-A	-	SME version 2.2. `sme` enables `sme2p2`. `sme2p2` enables `sme2p1`. `sme` and `sve2p2` enable `sme2p2`. See Scalable Matrix Extension for more information.	[BETA]
`sme-b16b16`	FEAT_SME_B16B16	Armv9.2-A	-	Non-widening BFloat16 to BFloat16 SME ZA-targeting arithmetic. `sme-b16b16` enables `sme2` and `sve-b16b16`.	Supported
`sme-i16i64`	FEAT_SME_I16I64	Armv9.2-A	-	16-bit to 64-bit integer widening outer product instructions. See Scalable Matrix Extension for more information.	Supported
`sme-f64f64`	FEAT_SME_F64F64	Armv9.2-A	-	Double-precision floating-point outer product instructions. See Scalable Matrix Extension for more information.	Supported
`sme-f16f16`	FEAT_SME_F16F16	Armv9.2-A	-	Non-widening half-precision FP16 to FP16 arithmetic for SME version 2.1. See Scalable Matrix Extension for more information.	Supported
`sme-f8f16`	FEAT_SME_F8F16	Armv9.2-A	-	SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`sme-f8f32`	FEAT_SME_F8F32	Armv9.2-A	-	SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`sme-lutv2`	FEAT_SME_LUTv2	Armv9.2-A	-	Additional SME2 lookup table instructions.	Supported
`ssbs`	FEAT_SSBS FEAT_SSBS2	Armv8-A	Armv8.5-A	Speculative Store Bypass Safe instructions. See Prevention of Speculative execution and data prediction for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`ssve-aes`	FEAT_SSVE_AES	Armv9.5-A	-	Streaming SVE Mode Advanced Encryption Standard and 128-bit polynomial multiply long instructions. `ssve-aes` enables `sme2p1` and `sve-pmull128`.	[BETA]
`ssve-f8f16mm`	FEAT_SSVE_F8F16MM	Armv9.2-A	-	Streaming SVE FP8 to half-precision Matrix Multiplication. `ssve-f8f16mm` enables `ssve-fp8dot2` and `ssve-f8f32mm`.	[BETA]
`ssve-f8f32mm`	FEAT_SSVE_F8F32MM	Armv9.2-A	-	Streaming SVE FP8 to single-precision Matrix Multiplication. `ssve-f8f32mm` enables `ssve-fp8dot4` and `sme-fsve`.	[BETA]
`ssve-fp8dot2`	FEAT_SSVE_FP8DOT2	Armv9.2-A	-	SVE2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats in Streaming SVE mode.	Supported
`ssve-fp8dot4`	FEAT_SSVE_FP8DOT4	Armv9.2-A	-	SVE2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats in Streaming SVE mode.	Supported
`ssve-fp8fma`	FEAT_SSVE_FP8FMA	Armv9.2-A	-	SVE2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats in Streaming SVE mode.	Supported
`sve`	FEAT_SVE	Armv8.2-A	-	*Scalable Vector Extension* (SVE). See Scalable Vector Extension for more information.	Supported
`sve2`	FEAT_SVE2	Armv9-A	-	SVE version 2. See Scalable Vector Extension for more information.	Supported
`sve2p1`	FEAT_SVE2p1	Armv9.4-A	-	SVE version 2.1. See Scalable Vector Extension for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`sve2p2`	FEAT_SVE2p2	Armv9.5-A	-	SVE version 2.2. `sve2` enables `sve2p2`. `sve2p2` enables `sve2p1`. See Scalable Vector Extension for more information.	[BETA]
`sve-aes2`	FEAT_SVE_AES2	Armv9.5-A	-	SVE multi-vector Advanced Encryption Standard and 128-bit polynomial multiply long instructions. `sve-aes2` enables `sve-pmull128`.	[BETA]
`sve-b16b16`	FEAT_SVE_B16B16	Armv9.2-A	-	Non-widening BFloat16 to BFloat16 arithmetic for SVE 2 and SME 2.	Supported
`sve-bfscale`	FEAT_SVE_BFSCALE	Armv9.2-A	-	BFloat16 floating-point adjust exponent. `sve-bfscale` enables `sve-b16b16`.	[BETA]
`sve-f16f32mm`	FEAT_SVE_F16F32MM	Armv9.2-A	-	SVE half-precision to single-precision Matrix Multiplication. `sve-f16f32mm` enables `sve2p1` or `sme-fsve`.	[BETA]
`the`	FEAT_THE	Armv8.9-A Armv9.4-A	-	Translation Hardening Extension. This extension provides additional instructions for updating page tables subject to user-space (EL0) and kernel-space (EL1) restrictions. Some `the` instructions require `d128` to be enabled.	Supported
`tlbiw`	FEAT_TLBIW	Armv9.5-A	-	Support for `TLBI` instructions that remove the stage 2 dirty state from *Translation Lookaside Buffer* (TLB) entries.	Supported
`tme`	FEAT_TME	Armv9-A	-	*Transactional Memory Extension* (TME). See Transactional Memory Extension for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

For targets in AArch64 state, the *Realm Management Extension* (RME) does not have an associated `+[no]<feature>` option. The RME registers are available in the Armv9-A application profile architecture without an additional extension.

Table 2-14: AArch64 state R-profile features

Command line `<feature>` option	Feature identifier	Description	Support level
`aes`	FEAT_AES FEAT_PMULL	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`crc`	FEAT_CRC32	CRC extension.	Supported
`dotprod`	FEAT_DotProd	Advanced SIMD dot product instructions. Enables the `SDOT` and `UDOT` instructions.	Supported
`flagm`	FEAT_FlagM	Flag Manipulation instructions.	Supported
`fp`	FEAT_FP	Floating-point extension. See Floating-point extensions for more information.	Supported. `+nofp` is only supported for C, not C++.
`fp16`	FEAT_FP16	Half-precision Floating-point extension. See Floating-point extensions for more information.	Supported
`fp16fml`	FEAT_FHM	Half-precision floating-point multiply with add or multiply with subtract extension. See Floating-point extensions for more information.	Supported
`pauth`	FEAT_PAuth	Pointer authentication extension. You can set this feature when using -mbranch-protection.	Supported
`pmuv3`	FEAT_PMUv3	*Performance Monitor Extension v3* (PMUv3). This option is enabled by default if you use `-mcpu` to enable code generation for a specific CPU that implements PMUv3. However, if you use `-march` to target an architecture profile, then `pmuv3` is not enabled by default and must be specified explicitly when required. This option only relates to the code generated by the compiler. The assembler always accepts PMUv3 system registers.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Description	Support level
`predres`	FEAT_SPECRES	Speculation restriction instructions. Enable instructions to prevent data prediction. See Prevention of Speculative execution and data prediction for more information.	Supported
`ras`	FEAT_RAS FEAT_RASv1p1	Reliability, Availability, and Serviceability extension.	Supported
`rcpc`	FEAT_LRCPC	Release Consistent Processor Consistent extension.	Supported
`sb`	FEAT_SB	Speculation barrier instruction. See Prevention of Speculative execution and data prediction for more information.	Supported
`sha2`	FEAT_SHA256 FEAT_SHA1	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`sha3`	FEAT_SHA3 FEAT_SHA512	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`simd`	FEAT_AdvSIMD	Advanced SIMD extension.	Supported. `+nosimd` is only supported for C, not C++.
`sm4`	FEAT_SM4	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`ssbs`	FEAT_SSBS FEAT_SSBS2	Speculative Store Bypass Safe instructions. See Prevention of Speculative execution and data prediction for more information.	Supported


Note

An AArch64 state R-profile feature is enabled by default or optional depending on whether the feature is enabled by default or optional in the Armv8.4-A architecture.

Table 2-15: AArch32 state A-profile features

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`aes`	FEAT_AES FEAT_PMULL	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`bf16`	FEAT_AA32BF16	Armv8.2-A	Armv8.6-A	BFloat16 Floating-point extension. See Floating-point extensions for more information.	Supported
`crc`	FEAT_CRC32	Armv8-A	Armv8.1-A	CRC extension.	Supported
`crypto`	FEAT_AES FEAT_PMULL FEAT_SHA1 FEAT_SHA256	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`dotprod`	FEAT_DotProd	Armv8.2-A	Armv8.4-A	Advanced SIMD dot product instructions. Enables the `VSDOT` and `VUDOT` instructions.	Supported
`fp16`	FEAT_FP16	Armv8.2-A	-	Half-precision Floating-point extension. See Floating-point extensions for more information.	Supported
`fp16fml`	FEAT_FHM	Armv8.2-A	-	Half-precision floating-point multiply with add or multiply with subtract extension. See Floating-point extensions for more information.	Supported
`i8mm`	FEAT_AA32I8MM	Armv8.2-A	Armv8.6-A Armv9.1-A	Matrix Multiply extension. See Matrix Multiplication extension for more information.	Supported
`ras`	FEAT_RAS FEAT_RASv1p1	Armv8-A	Armv8.2-A	Reliability, Availability, and Serviceability extension.	Supported
`sb`	FEAT_SB	Armv8-A	Armv8.5-A	Speculation barrier instruction. See Prevention of Speculative execution and data prediction for more information.	Supported
`sha2`	FEAT_SHA256 FEAT_SHA1	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

For targets in AArch32 state, you can use `-mfpu` to specify the support for floating-point, Advanced SIMD, and Cryptographic extensions.

Table 2-16: AArch32 state M-profile features

Command line `<feature>` option	Optional from	Enabled by default	Description	Support level
`cdecp<N>`	Armv8-M	-	*Custom Datapath Extension* (CDE). `<N>` is in the range 0-7. See Custom Datapath Extension for more information.	Supported
`dsp`	Armv8-M	-	*Digital Signal Processing* (DSP) extension for the Armv8-M.mainline architecture.	Supported
`lob`	Armv8.1-M	Armv8.1-M	Low Overhead Branch extension.	Supported
`mve`	Armv8.1-M	-	*M-Profile Vector Extension* (MVE). See M-profile Vector Extension for more information.	Supported
`pacbti`	Armv8.1-M	-	*Pointer Authentication and Branch Target Identification* (PACBTI) extension. See Armv8.1-M PACBTI extension for more information.	Supported

For architecture extensions supported by Armv8-R AArch32, see Supported architecture features.

AArch64 A-profile

`armlink` uses AArch64 library code that can contain floating-point and Advanced SIMD instructions and registers. The use of the AArch64 libraries applies even if you compile the source with `-march=<name>+nofp+nosimd` to prevent the compiler from using floating-point and SIMD instructions and registers. Therefore, there is no guarantee that the linked image for targets in AArch64 state is entirely free of Advanced SIMD and floating-point instructions and registers.

You can prevent the use of Advanced SIMD and floating-point instructions and registers in images that are linked for targets in AArch64 state. Either reimplement the library functions or create your own library that does not use Advanced SIMD and floating-point instructions and registers.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

AArch64 R-profile

`armlink` links against AArch64 library code that can use floating-point and SIMD instructions and registers. To use library variants that do not contain floating-point instructions compile with the `+nofp` architecture feature.

Example: Use of an invalid feature with the Armv8-A architecture

Create the file `test_fadd.s` containing the following assembly code:

```
    .section .text.test_fadd, "ax"
    .balign 8
    .global test_fadd
    .type test_fadd, %function
test_fadd:
    fadd v0.4h, v0.4h, v1.4h
    ret
```

Assemble `test_fadd.s` with the following command:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a+fp16 -Wl,--entry=test_fadd
 test_fadd.s
```

`+fp16` enables the Half-precision floating-point data-processing feature, FEAT_FP16. However, this feature is only supported in Armv8.2-A and later.

The integrated assembler does not report an error or a warning for the half-precision floating-point `FADD` instruction in this code when assembling with `-march=armv8-a+fp16`.

Subsequently, the combination `-march=armv8-a+fp16` can result in unexpected run-time behavior when the `FADD` instruction is executed on a target that does not support FEAT_FP16.

Related information

-mcpu on page 150
-marm on page 140
-mthumb on page 193
--target on page 221
--coprocN=value (fromelf) on page 776
Half-precision floating-point data types on page 342
Half-precision floating-point intrinsics on page 347
Overview of the Realm Management Extension
Custom Datapath Extension support

2.1.70 -marm

Requests that the compiler targets the A32 instruction set.

Default

The default for all targets that support A32 instructions is `-marm`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
-marm
```

Parameters

None.

Operation

Different architectures support different instruction sets:

- Arm®v8-A and Armv9-A processors in AArch64 state execute A64 instructions.
- Armv8-A and Armv9-A processors in AArch32 state, in addition to Armv7 and earlier A- and R-profile processors execute A32 and T32 instructions.
- M-profile processors execute T32 instructions.



Note

This option is only valid for targets that support the A32 instruction set.

- The compiler generates an error and stops if the `-marm` option is used with an M-profile target architecture. No code is produced.
- The compiler ignores the `-marm` option and generates a warning if used with AArch64 targets.

Related information

-mthumb on page 193
--target on page 221
-mcpu on page 150

2.1.71 -masm

Enables Arm® Development Studio to select the correct assembler for the input assembly source files.

Default

The default is `-masm=gnu`.

Syntax

```
-masm=<assembler>
```

Parameters

`<assembler>`

One of the following:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

auto

Automatically detect the correct assembler from the syntax of the input assembly source file.

If the assembly source file contains GNU syntax assembly code, then invoke `armclang` integrated assembler.

If the assembly source file contains legacy `armasm` syntax assembly code, then invoke the legacy `armasm` assembler. The most commonly used options are translated from the `armclang` command-line options to the appropriate `armasm` command-line options.


Note

In rare circumstances, the auto-detection might select the wrong assembler for the input file. For example, if the input file is a `.s` file that requires preprocessing, auto-detection might select the wrong assembler. For the files where auto-detection selects the wrong assembler, you must select `-masm=gnu` or `-masm=armasm` explicitly.

gnu

Invoke the `armclang` integrated assembler.

armasm

Invoke the legacy `armasm` assembler. The most commonly used options are translated from the `armclang` command-line options to the appropriate `armasm` command-line options.

Operation

If you use Arm Development Studio to build projects with the CMSIS-Pack, use `-masm=auto`, because some of the assembly files in the CMSIS-Pack contain legacy `armasm` syntax assembly code. When invoking the legacy `armasm` assembler, the most commonly used options are translated from the `armclang` command-line options to the appropriate `armasm` command-line options, which the Translatable options table shows.

Table 2-17: Translatable options

armclang option	armasm option
`-mcpu, -march`	`--cpu`
`-marm`	`--arm`
`-mthumb`	`--thumb`
`-fropi`	`--apcs=/ropi`
`-frwpi`	`--apcs=/rwpi`

`-mfloat-abi=soft`
 	
`-mfloat-abi=softfp`
 	
`-mfloat-abi=hard`
	`--apcs=/softfp`
	
	`--apcs=/softfp`
	
	`--apcs=/hardfp`

`-mfpu`	`--fpu`
`-mbig-endian`	`--bigend`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

armclang option	armasm option
`-mlittle-endian`	`--littleend`
`-g`	`-g`
`-ffp-mode`	`--fpmode`
`-DNAME`	`--predefine "NAME SETA 1"`
`-I<dir>`	`-i<dir>`

If you need to provide additional options to the legacy `armasm` assembler, which are not listed in the Translatable options table, then use `-Wa,armasm,<option>,<value>`. For example:

- If you want to use the legacy `armasm` assembler option `--show_cmdline` to see the command-line options that have been passed to the legacy `armasm` assembler, then use:

```
-Wa,armasm,--show_cmdline
```

- If the legacy `armasm` syntax source file requires the option `--predefine "NAME SETA 100"`, then use:

```
-Wa,armasm,--predefine,"NAME SETA 100"
```

- If the legacy `armasm` syntax source file requires the option `--predefine "NAME SETS \"Version 1.0\""`, then use:

```
-Wa,armasm,--predefine,"NAME SETS \"Version 1.0\""
```

Note

The command-line interface of your system might require you to enter special character combinations to achieve correct quoting, such as `\\"` instead of `"`.

Note

Arm Compiler for Embedded 6 provides the `-masm` option as a short term solution to enable the assembly of legacy `armasm` syntax assembly source files. The `-masm` option will be removed in a future release of Arm Compiler for Embedded 6.

We recommend that you migrate all legacy `armasm` syntax assembly source files into GNU syntax assembly source files. For more information, see Migrating from armasm to the armclang Integrated Assembler in the Migration and Compatibility Guide.

If you are using the compiler from outside Arm Development Studio, such as from the command-line, then we recommend that you do not specify the `-masm` option, and instead invoke the correct assembler explicitly.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.72 -mbig-endian

Generates code suitable for an Arm® processor using big-endian memory.

Default

The default is `-mlittle-endian`.

Syntax

```
-mbig-endian
```

Parameters

None.

Operation

The Arm architecture defines the following big-endian modes:

BE-8

Byte-invariant addressing mode (Arm®v6 and later).

BE-32

Word-invariant addressing big-endian mode.

The selection of BE-8 or BE-32 is specified at link time.

Related information

--be8 on page 431
--be32 on page 431
-mlittle-endian on page 183

2.1.73 -mbranch-protection

Controls the level and type of branch protection. This option is valid for AArch64 state, and for Arm®v8-M targets.

Default

The default is `-mbranch-protection=none`.

Syntax

```
-mbranch-protection=<protection>[+<protection>+...]
```

Parameters

`<protection>`

The level or type of protection. One or more of the following values:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

none

Disables all types of branch protection.

standard

Enables *Branch Target Identification* (BTI) and *Pointer Authentication Code* (PAC) branch protection based on the specified `-march=<name>` and `-mcpu=<name>` target options. For more information, see Example: -mbranch-protection=standard.

bti

Enables branch protection using BTI. For more information, see Example: -mbranch-protection=bti.

pac-ret[+<option>+...]

Enables branch protection using PAC. If you use `pac-ret` protection, you can specify additional options using the `+` separator.

Options for AArch64 state:

b-key

Performs pointer authentication using PAC Key B instead of PAC Key A. For more information, see How is the PAC formed?.

leaf

Enables pointer authentication for all functions, including leaf functions. A leaf function is a function that does not save the *Link Register* (LR) on the stack.

pc

Performs pointer authentication using the *Program Counter* (PC) as the second diversifier for return address signing in addition to SP as the first diversifier. A diversifier is also known as a modifier. For more information, see What is the modifier?.

Options for Armv8-M targets with the Main Extension:

leaf

Enables pointer authentication for all functions, including leaf functions. A leaf function is a function that does not save the LR on the stack.


Note

You can specify multiple `+<option>` parameters. Include these options immediately after `pac-ret`. For example, `-mbranch-protection=pac-ret+leaf+pc` is valid, but `-mbranch-protection=leaf+pac-ret+pc` is invalid.

For more information, see Example: -mbranch-protection=pac-ret.

Operation

Depending on the `<protection>` levels or types specified in `-mbranch-protection=<protection>`, the compiler can add branch protection instructions to the code it generates for the functions in a source file.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The branch protection instructions added for each function in a source file depend on the following factors:

- Whether the specified target options indicate that the target includes hardware support for branch protection features.
- Whether a function is a leaf function that saves the *Link Register* (LR) on the stack.
- Whether the function is annotated with the `__attribute__((target("branch-protection=<protection>")))` function attribute.

For all valid targets, you can use `-mbranch-protection` in combination with:

- `+<feature>` modifiers for the `-march=<name>` and `-mcpu=<name>` options that control the type of instructions used to apply branch protection.
- The `__attribute__((target("branch-protection=<protection>")))` function attribute. For more information, see __attribute__((target("options"))) function attribute.

For Armv8-M targets, you can use `-mbranch-protection` in combination with the following options:

- Linker options that control whether branch protection is also applied to the unqualified C libraries supplied with the toolchain. For more information, see --library_security=protection.
- Linker options that control whether the linker reports an error, or a warning, if branch protection is not applied consistently to all input objects. For more information, see --require-bti and --check_pac_mismatch.

Example: -mbranch-protection examples

The `-mbranch-protection` examples include more information about `standard`, `bti`, and `pac-ret` protection.

The examples are available in Examples for the armclang -mbranch-protection command-line option.

Related information

-march on page 123

2.1.74 -mcmodel

Selects the generated code model.

Default

The default is `-mcmodel=small`.

Syntax

```
-mcmodel=<model>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<model>

Specifies the model type for code generation.

When specifying the model type, it can be one of:

tiny

Generate code for the tiny code model. The program and its statically defined symbols must be within 1MB of each other.

small

Generate code for the small code model. The program and its statically defined symbols must be within 4GB of each other. This is the default code model.

large

Generate code for the large code model. The compiler makes no assumptions about addresses and sizes of sections.



Use the large code model for programs with large amounts of read-write data, not large amounts or sparsely placed code.

None of the libraries in the Arm® Compiler for Embedded toolchain have been compiled using the large code model.

Restrictions

All code models are supported on AArch64 only.

Operation

The compiler generates instructions that refer to global data through relative offset addresses. The model type specifies the maximum offset range, and therefore the size of the offset address. The actual required range of an offset is only known when the program is linked with other object files and libraries. If you know the final size of your program, you can specify the appropriate model so that the compiler generates optimal code for offsets.

If you specify a larger model than is required, then your code is unnecessarily larger.

If the model you choose is too small and the image does not fit in the bounds, then the linker reports an error.

Examples

This example enables code generation for the large code model:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -mcmodel=large -c foo.c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

This example generates code for the default (small) code model:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -c foo.c
```

Related information

Linker-generated veneers on page 597

2.1.75 -mcmse

Enables the generation of code for the Secure state of the Arm®v8-M Security Extension (CMSE). This option is required when creating a Secure image.

Syntax

```
-mcmse
```

Parameters

None.

Restrictions

- The Armv8-M Security Extension is not supported when building *Read-Only Position Independent* (ROPI) and *Read/Write Position Independent* (RWPI) images.
- Mixing objects compiled for Armv8-M.baseline and Armv8-M.mainline could potentially leak sensitive data, because Armv8-M.baseline does not support the Floating-Point Extension. Therefore, the compiler cannot generate code to clear the Secure floating-point registers when performing a Non-secure call. If any object is compiled for the Armv8-M.mainline architecture, all files containing CMSE attributes must be compiled for the Armv8-M.mainline architecture.

Operation

Specifying `-mcmse` targets the Secure state of the Armv8-M Security Extension. When compiling with `-mcmse`, the following are available:

- The Test Target, `TTA`, instruction.
- `TTA` instruction intrinsics.
- Non-secure function pointer intrinsics.
- `__attribute__((cmse_nonsecure_call))` and `__attribute__((cmse_nonsecure_entry))` function attributes.

Note

- The value of the predefined macro `__ARM_FEATURE_CMSE` indicates what Armv8-M Security Extension features are supported.
- Compile Secure code with the maximum capabilities for the target. For example, if you compile with no FPU then the Secure functions do not clear floating-point registers when returning from functions declared as `__attribute__((cmse_nonsecure_entry))`. Therefore, the functions could potentially leak sensitive data.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Structs with undefined bits caused by padding and half-precision floating-point members are currently unsupported as arguments and return values for Secure functions. Using such structs might leak sensitive information. Structs that are large enough to be passed by reference are also unsupported and produce an error.
- The following cases are not supported when compiling with the `armclang` option `-mcmse` and produce an error:
  - Variadic entry functions.
  - Entry functions with arguments that do not fit in registers, because there are either many arguments or the arguments have large values.
  - Non-secure function calls with arguments that do not fit in registers, because there are either many arguments or the arguments have large values.
- You might have more arguments in entry functions or Non-secure function calls than can fit in registers. In this situation, you can pass a pointer to a struct containing all the arguments. For example:

```
typedef struct {
    int p1;
    int p2;
    int p3;
    int p4;
    int p5;
} Params;

void your_api(int p1, int p2, int p3, int p4, int p5) {
    Params p1 = { p1, p2, p3, p4, p5 };
    your_api_implementation(&p1);
}
```

  Here, `your_api_implementation(&p1)` is the call to your existing function, with fewer than the maximum number of 4 arguments allowed.

Example

This example shows the command-line invocations for creating a Secure image using an input import library, `oldimportlib.o`:

```
armclang --target=arm-arm-none-eabi -march=armv8m.main -mcmse secure.c -o secure.o
armlink secure.o -o secure.axf --import-cmse-lib-out importlib.o --import-cmse-lib-
in oldimportlib.o --scatter secure.scat
```

`armlink` also generates the Secure code import library, `importlib.o` that is required for a Non-secure image to call the Secure image.

Related information

-march on page 123
-mfpu on page 170
--target on page 221

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

__attribute__((cmse_nonsecure_call)) function attribute on page 260
__attribute__((cmse_nonsecure_entry)) function attribute on page 261
Predefined macros on page 336
TT instruction intrinsics on page 350
Non-secure function pointer intrinsics on page 353
--fpu=name (armlink) on page 470
--import_cmse_lib_in=filename on page 472
--import_cmse_lib_out=filename on page 473
--scatter=filename on page 529
Building Secure and Non-secure Images Using the Armv8-M Security Extension
TT, TTT, TTA, TTAT

2.1.76 -mcpu

Enables code generation for a specific Arm® processor.


Note

This topic includes descriptions of [ALPHA] features. See Support level definitions.


Note

Avoid specifying both the architecture (`-march`) and the processor (`-mcpu`) because specifying both has the potential to cause a conflict. The compiler infers the correct architecture from the processor.

Default

There is no default `-mcpu` option.

When compiling with `--target=aarch64-arm-none-eabi`, the default is `-march=armv8-a`.

When compiling with `--target=arm-arm-none-eabi`, the default is unsupported. Ensure that you always use one of the `-march` or `-mcpu` options when compiling with `--target=arm-arm-none-eabi`.


Note

When compiling with `--target=arm-arm-none-eabi` and without `-march` or `-mcpu`, the compiler generates code that is not compatible with any supported architectures or processors, and reports:

```
warning: 'armv4t' is unsupported in this version of the product
```

To see the default floating-point configuration for your processor:

1. Compile with `-mcpu=<name> -S` to generate the assembler file.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2. Open the assembler file and check that the value for the `.fpu` directive corresponds to one of the `-mfpu` options. No `.fpu` directive implies `-mfpu=none`.

Syntax

To specify a target processor:

```
-mcpu=<name>
```

To enable or disable optional extensions supported by a processor:

```
-mcpu=<name>[+[no]<feature>+...]
```

To view a list of all the supported processors:

```
-mcpu=list
```

Parameters

`<name>`

The name of the processor.

`<feature>`

An optional architecture feature that might be enabled or disabled by default depending on the architecture or processor.



The compiler and integrated assembler do not prevent the use of `+<feature>` modifiers with an incompatible target. `armclang` is not guaranteed to report an error or warning for invalid architectural feature combinations. Therefore, you must manually ensure that you only specify valid `+<feature>` and `+no<feature>` modifiers for the `-march` and `-mcpu` options.



- If a feature is mandatory in an architecture, then that feature is enabled by default. For AArch32 state inputs only, you can use `fromelf --decode_build_attributes` to determine whether the feature is enabled.
- For some Arm processors, `-mcpu` supports specific combinations of the architecture features. See Supported architecture feature combinations for specific processors for more information.

`+<feature>` enables the feature if it is disabled by default. `+<feature>` has no effect if the feature is already enabled by default.

`+no<feature>` disables the feature if it is enabled by default. `+no<feature>` has no effect if the feature is already disabled by default.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Use `+<feature>` or `+no<feature>` to explicitly enable or disable an optional architecture feature.

You can specify one or more features that an architecture supports.

Table 2-18: AArch64 state A-profile features

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`aes`	FEAT_AES FEAT_PMULL	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`b16b16`	FEAT_SVE_B16B16	Armv9-A	-	SVE2.1 and SME2.1 non-widening BFloat16 instructions. See Scalable Matrix Extension for more information.	[ALPHA]
`bf16`	FEAT_BF16	Armv8.2-A	Armv8.6-A	BFloat16 Floating-point extension. See Floating-point extensions for more information.	Supported
`brbe`	FEAT_BRBE	Armv9.2-A	-	Invalidate the Branch Record Buffer extension. Enables support for the `BRB IALL` and `BRB INJ` instructions and system registers.	Supported
`cmpbr`	FEAT_CMPBR	Armv9.5-A	Armv9.6-A	Compare and Branch instructions.	[BETA]
`cpa`	FEAT_CPA	Armv9.4-A	Armv9.5-A	*Checked Pointer Arithmetic* (CPA) extension.	Supported
`crc`	FEAT_CRC32	Armv8-A	Armv8.1-A	*Cyclic Redundancy Check* (CRC) extension.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`crypto`	FEAT_AES FEAT_PMULL FEAT_SHA1 FEAT_SHA256 Armv8.4-A and later also includes: FEAT_SHA512 FEAT_SHA3 FEAT_SM3 FEAT_SM4	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`cssc`	FEAT_CSSC	Armv8.7-A	Armv8.9-A Armv9.4-A	*Common Short Sequence Compression* (CSSC) instructions.	Supported
`d128`	FEAT_D128	Armv9.4-A	-	128-bit page table descriptor. `d128` enables FEAT_D128 but also FEAT_SYSREG128 and FEAT_SYSINSTR128. `d128` also enables the `lse128` feature, FEAT_LSE128.	Supported
`dotprod`	FEAT_DotProd	Armv8.2-A	Armv8.4-A	Advanced *Single Instruction Multiple Data* (SIMD) dot product instructions. Enables the `SDOT` and `UDOT` instructions.	Supported
`f32mm`	FEAT_F32MM	Armv8.2-A	-	Matrix Multiply extension. See Matrix Multiplication extension for more information.	Supported
`f64mm`	FEAT_F64MM	Armv8.2-A	-	Matrix Multiply extension. See Matrix Multiplication extension for more information.	Supported
`f8f16mm`	FEAT_F8F16MM	Armv9.2-A	-	FP8 to half-precision Matrix Multiplication. `f8f16mm` enables `f8f32mm` and `fp8dot2`. See Matrix Multiplication extension for more information.	[BETA]

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`f8f32mm`	FEAT_F8F32MM	Armv9.2-A	-	FP8 to single-precision Matrix Multiplication. `f8f32mm` enables `fp8dot4`. See Matrix Multiplication extension for more information.	[BETA]
`faminmax`	FEAT_FAMINMAX	Armv9.5-A	Armv9.5-A	Floating-point absolute minimum and maximum. Advanced SIMD, SVE2, and SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`flagm`	FEAT_FlagM	Armv8.2-A	Armv8.4-A	Flag Manipulation instructions.	Supported
`fp`	FEAT_FP	Armv8-A	-	Floating-point extension. See Floating-point extensions for more information.	Supported
`fp8`	FEAT_FP8	Armv9.2-A	-	Advanced SIMD, SVE2, and SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`fp8dot2`	FEAT_FP8DOT2	Armv9.2-A	-	Advanced SIMD, SVE2, and SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`fp8dot4`	FEAT_FP8DOT4	Armv9.2-A	-	Advanced SIMD, SVE2, and SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`fp8fma`	FEAT_FP8FMA	Armv9.2-A	-	Advanced SIMD, SVE2, and SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`fp16`	FEAT_FP16	Armv8.2-A	-	Half-precision Floating-point extension. See Floating-point extensions for more information.	Supported
`fp16fml`	FEAT_FHM	Armv8.2-A	-	Half-precision floating-point multiply with add or multiply with subtract extension. See Floating-point extensions for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`fprcvt`	FEAT_FPRCVT	Armv9.5-A	Armv9.6-A	Floating-point to/from integer in scalar FP register.	[BETA]
`gcs`	FEAT_GCS	Armv9.4-A	-	The *Guarded Control Stack* (GCS) extension enables you to securely store return addresses, and provides a method to check that a return address is not compromised.	[ALPHA]
`hbc`	FEAT_HBC	-	Armv8.8-A Armv9.3-A	Hinted Conditional Branches extension. This extension is disabled by default for the Armv8.7-A and Armv9.2-A architectures.	Supported
`i8mm`	FEAT_I8MM	Armv8.2-A	Armv8.6-A Armv9.1-A	Matrix Multiply extension. See Matrix Multiplication extension for more information.	Supported
`ite`	FEAT_ITE	Armv9.4-A	-	Instrumentation Extension. This extension enables a `TRCIT xN` instruction for software to add data into the ETE trace stream.	Supported
`ls64`	FEAT_LS64 FEAT_LS64_V FEAT_LS64_ACCDATA	Armv8.7-A	-	Accelerator support extension for 64-byte atomic loads and stores.	Supported
`lse128`	FEAT_LSE128	Armv9.4-A	-	128-bit atomic instructions. Provides instructions that can atomically read-modify-write 128 bits of a memory location. 128-bit read-modify-write of a memory location is required for editing the 128-bit translation table descriptor.	Supported
`lsfe`	FEAT_LSFE	Armv9.5-A	-	Large System Float Extension.	[BETA]
`lsui`	FEAT_LSUI	Armv9.5-A	Armv9.6-A	Unprivileged load store instructions.	[BETA]

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`lut`	FEAT_LUT	Armv9.2-A	Armv9.5-A	Lookup-table instructions for Advanced SIMD and optionally SVE2 and SME2 if those extensions are implemented.	Supported
`memtag`	FEAT_MTE FEAT_MTE2	Armv8.5-A	-	Memory tagging extension. See Overview of memory tagging for more information.	Supported
`mops`	FEAT_MOPS	-	Armv8.8-A Armv9.3-A	Memory Operations extension. Adds support for instructions to standardize the `memcpy`, `memmove`, and `memset` operations. The extension is disabled by default for the Armv8.7-A and Armv9.2-A architectures. See A-profile Memory Operations Extension for more information.	Supported
`occmo`	FEAT_OCCMO	Armv9.5-A	Armv9.6-A	Outer Cacheable Cache Maintenance Operation.	[BETA]
`pauth`	FEAT_PAuth	Armv8.2-A	Armv8.3-A	Pointer authentication extension. You can set this feature when using -mbranch-protection to ensure pointer authentication works in the same way on Armv8.2-A architectures as it does on Armv8.3-A and later architectures.	Supported
`pauth-lr`	FEAT_PAuth_LR	Armv9.5-A	-	Pointer authentication instructions that enable signing of the *Link Register* (LR) using the *Stack Pointer* (SP) and the *Program Counter* (PC) as diversifiers. See -mbranch-protection for more information.	Supported
`pcdphint`	FEAT_PCDPHINT	Armv9.5-A	-	Producer-Consumer Data Placement Hints	[BETA]

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`pmuv3`	FEAT_PMUv3	Armv8-A	-	*Performance Monitor Extension v3* (PMUv3). This option is enabled by default if you use `-mcpu` to enable code generation for a specific CPU that implements PMUv3. However, if you use `-march` to target an architecture profile, then `pmuv3` is not enabled by default and must be specified explicitly when required. This option only relates to the code generated by the compiler. The assembler always accepts PMUv3 system registers.	Supported
`pops`	FEAT_PoPS	Armv9.5-A	-	Point of Physical Storage	[BETA]
`predres`	FEAT_SPECRES	Armv8-A	Armv8.5-A	Speculation restriction instructions. Enable instructions to prevent data prediction. See Prevention of Speculative execution and data prediction for more information.	Supported
`predres2`	FEAT_SPECRES2	Armv8-A	Armv8.9-A Armv9.4-A	Enhanced speculation restriction instructions. See Prevention of Speculative execution and data prediction for more information.	Supported
`profile`	FEAT_SPE	Armv8.2-A	-	Statistical Profiling Extension.	Supported
`ras`	FEAT_RAS FEAT_RASv1p1	Armv8-A	Armv8.2-A	Reliability, Availability, and Serviceability extension.	Supported
`rasv2`	FEAT_RASv2	-	Armv8.9-A Armv9.4-A	Reliability, Availability, and Serviceability extension version 2. `rasv2` adds an additional system register to `ras`.	Supported
`rcpc`	FEAT_LRCPC	-	Armv8.3-A	Release Consistent Processor Consistent extension.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`rcpc3`	FEAT_LRCPC3	Armv8.2-A	-	Additional support for the Release Consistent Processor Consistent extension. `rcpc3` also enables the `rcpc` feature.	Supported
`rme-gpc3`	FEAT_RME_GPC3	Armv9.5-A	-	RME Granule Protection Check 3 extension. `rme-gpc3` enables `rme-gpc` and `rme-gpc2`.	[BETA]
`rng`	FEAT_RNG	Armv8.5-A	-	Random Number Generation extension.	Supported
`sb`	FEAT_SB	Armv8-A	Armv8.5-A	Speculation barrier instruction. See Prevention of Speculative execution and data prediction for more information.	Supported
`sha2`	FEAT_SHA256 FEAT_SHA1	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`sha3`	FEAT_SHA3 FEAT_SHA512	Armv8.2-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`simd`	FEAT_AdvSIMD	Armv8-A	-	Advanced SIMD extension.	Supported
`sm4`	FEAT_SM4	Armv8.2-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`sme`	FEAT_SME	Armv9.2-A	-	*Scalable Matrix Extension* (SME). See Scalable Matrix Extension for more information.	Supported
`sme2p1`	FEAT_SME2p1	Armv9.2-A	-	SME version 2.1. See Scalable Matrix Extension for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`sme2p2`	FEAT_SME2p2	Armv9.5-A	-	SME version 2.2. `sme` enables `sme2p2`. `sme2p2` enables `sme2p1`. `sme` and `sve2p2` enable `sme2p2`. See Scalable Matrix Extension for more information.	[BETA]
`sme-b16b16`	FEAT_SME_B16B16	Armv9.2-A	-	Non-widening BFloat16 to BFloat16 SME ZA-targeting arithmetic. `sme-b16b16` enables `sme2` and `sve-b16b16`.	Supported
`sme-i16i64`	FEAT_SME_I16I64	Armv9.2-A	-	16-bit to 64-bit integer widening outer product instructions. See Scalable Matrix Extension for more information.	Supported
`sme-f64f64`	FEAT_SME_F64F64	Armv9.2-A	-	Double-precision floating-point outer product instructions. See Scalable Matrix Extension for more information.	Supported
`sme-f16f16`	FEAT_SME_F16F16	Armv9.2-A	-	Non-widening half-precision FP16 to FP16 arithmetic for SME version 2.1. See Scalable Matrix Extension for more information.	Supported
`sme-f8f16`	FEAT_SME_F8F16	Armv9.2-A	-	SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`sme-f8f32`	FEAT_SME_F8F32	Armv9.2-A	-	SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats.	Supported
`sme-lutv2`	FEAT_SME_LUTv2	Armv9.2-A	-	Additional SME2 lookup table instructions.	Supported
`ssbs`	FEAT_SSBS FEAT_SSBS2	Armv8-A	Armv8.5-A	Speculative Store Bypass Safe instructions. See Prevention of Speculative execution and data prediction for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`ssve-aes`	FEAT_SSVE_AES	Armv9.5-A	-	Streaming SVE Mode Advanced Encryption Standard and 128-bit polynomial multiply long instructions. `ssve-aes` enables `sme2p1` and `sve-pmull128`.	[BETA]
`ssve-f8f16mm`	FEAT_SSVE_F8F16MM	Armv9.2-A	-	Streaming SVE FP8 to half-precision Matrix Multiplication. `ssve-f8f16mm` enables `ssve-fp8dot2` and `ssve-f8f32mm`.	[BETA]
`ssve-f8f32mm`	FEAT_SSVE_F8F32MM	Armv9.2-A	-	Streaming SVE FP8 to single-precision Matrix Multiplication. `ssve-f8f32mm` enables `ssve-fp8dot4` and `sme-fsve`.	[BETA]
`ssve-fp8dot2`	FEAT_SSVE_FP8DOT2	Armv9.2-A	-	SVE2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats in Streaming SVE mode.	Supported
`ssve-fp8dot4`	FEAT_SSVE_FP8DOT4	Armv9.2-A	-	SVE2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats in Streaming SVE mode.	Supported
`ssve-fp8fma`	FEAT_SSVE_FP8FMA	Armv9.2-A	-	SVE2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats in Streaming SVE mode.	Supported
`sve`	FEAT_SVE	Armv8.2-A	-	*Scalable Vector Extension* (SVE). See Scalable Vector Extension for more information.	Supported
`sve2`	FEAT_SVE2	Armv9-A	-	SVE version 2. See Scalable Vector Extension for more information.	Supported
`sve2p1`	FEAT_SVE2p1	Armv9.4-A	-	SVE version 2.1. See Scalable Vector Extension for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`sve2p2`	FEAT_SVE2p2	Armv9.5-A	-	SVE version 2.2. `sve2` enables `sve2p2`. `sve2p2` enables `sve2p1`. See Scalable Vector Extension for more information.	[BETA]
`sve-aes2`	FEAT_SVE_AES2	Armv9.5-A	-	SVE multi-vector Advanced Encryption Standard and 128-bit polynomial multiply long instructions. `sve-aes2` enables `sve-pmull128`.	[BETA]
`sve-b16b16`	FEAT_SVE_B16B16	Armv9.2-A	-	Non-widening BFloat16 to BFloat16 arithmetic for SVE 2 and SME 2.	Supported
`sve-bfscale`	FEAT_SVE_BFSCALE	Armv9.2-A	-	BFloat16 floating-point adjust exponent. `sve-bfscale` enables `sve-b16b16`.	[BETA]
`sve-f16f32mm`	FEAT_SVE_F16F32MM	Armv9.2-A	-	SVE half-precision to single-precision Matrix Multiplication. `sve-f16f32mm` enables `sve2p1` or `sme-fsve`.	[BETA]
`the`	FEAT_THE	Armv8.9-A Armv9.4-A	-	Translation Hardening Extension. This extension provides additional instructions for updating page tables subject to user-space (EL0) and kernel-space (EL1) restrictions. Some `the` instructions require `d128` to be enabled.	Supported
`tlbiw`	FEAT_TLBIW	Armv9.5-A	-	Support for `TLBI` instructions that remove the stage 2 dirty state from *Translation Lookaside Buffer* (TLB) entries.	Supported
`tme`	FEAT_TME	Armv9-A	-	*Transactional Memory Extension* (TME). See Transactional Memory Extension for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

For targets in AArch64 state, the *Realm Management Extension* (RME) does not have an associated `+[no]<feature>` option. The RME registers are available in the Armv9-A application profile architecture without an additional extension.

Table 2-19: AArch64 state R-profile features

Command line `<feature>` option	Feature identifier	Description	Support level
`aes`	FEAT_AES FEAT_PMULL	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`crc`	FEAT_CRC32	CRC extension.	Supported
`dotprod`	FEAT_DotProd	Advanced SIMD dot product instructions. Enables the `SDOT` and `UDOT` instructions.	Supported
`flagm`	FEAT_FlagM	Flag Manipulation instructions.	Supported
`fp`	FEAT_FP	Floating-point extension. See Floating-point extensions for more information.	Supported. `+nofp` is only supported for C, not C++.
`fp16`	FEAT_FP16	Half-precision Floating-point extension. See Floating-point extensions for more information.	Supported
`fp16fml`	FEAT_FHM	Half-precision floating-point multiply with add or multiply with subtract extension. See Floating-point extensions for more information.	Supported
`pauth`	FEAT_PAuth	Pointer authentication extension. You can set this feature when using -mbranch-protection.	Supported
`pmuv3`	FEAT_PMUv3	*Performance Monitor Extension v3* (PMUv3). This option is enabled by default if you use `-mcpu` to enable code generation for a specific CPU that implements PMUv3. However, if you use `-march` to target an architecture profile, then `pmuv3` is not enabled by default and must be specified explicitly when required. This option only relates to the code generated by the compiler. The assembler always accepts PMUv3 system registers.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Description	Support level
`predres`	FEAT_SPECRES	Speculation restriction instructions. Enable instructions to prevent data prediction. See Prevention of Speculative execution and data prediction for more information.	Supported
`ras`	FEAT_RAS FEAT_RASv1p1	Reliability, Availability, and Serviceability extension.	Supported
`rcpc`	FEAT_LRCPC	Release Consistent Processor Consistent extension.	Supported
`sb`	FEAT_SB	Speculation barrier instruction. See Prevention of Speculative execution and data prediction for more information.	Supported
`sha2`	FEAT_SHA256 FEAT_SHA1	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`sha3`	FEAT_SHA3 FEAT_SHA512	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`simd`	FEAT_AdvSIMD	Advanced SIMD extension.	Supported. `+nosimd` is only supported for C, not C++.
`sm4`	FEAT_SM4	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`ssbs`	FEAT_SSBS FEAT_SSBS2	Speculative Store Bypass Safe instructions. See Prevention of Speculative execution and data prediction for more information.	Supported


Note

An AArch64 state R-profile feature is enabled by default or optional depending on whether the feature is enabled by default or optional in the Armv8.4-A architecture.

Table 2-20: AArch32 state A-profile features

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`aes`	FEAT_AES FEAT_PMULL	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Command line `<feature>` option	Feature identifier	Optional from	Enabled by default	Description	Support level
`bf16`	FEAT_AA32BF16	Armv8.2-A	Armv8.6-A	BFloat16 Floating-point extension. See Floating-point extensions for more information.	Supported
`crc`	FEAT_CRC32	Armv8-A	Armv8.1-A	CRC extension.	Supported
`crypto`	FEAT_AES FEAT_PMULL FEAT_SHA1 FEAT_SHA256	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported
`dotprod`	FEAT_DotProd	Armv8.2-A	Armv8.4-A	Advanced SIMD dot product instructions. Enables the `VSDOT` and `VUDOT` instructions.	Supported
`fp16`	FEAT_FP16	Armv8.2-A	-	Half-precision Floating-point extension. See Floating-point extensions for more information.	Supported
`fp16fml`	FEAT_FHM	Armv8.2-A	-	Half-precision floating-point multiply with add or multiply with subtract extension. See Floating-point extensions for more information.	Supported
`i8mm`	FEAT_AA32I8MM	Armv8.2-A	Armv8.6-A Armv9.1-A	Matrix Multiply extension. See Matrix Multiplication extension for more information.	Supported
`ras`	FEAT_RAS FEAT_RASv1p1	Armv8-A	Armv8.2-A	Reliability, Availability, and Serviceability extension.	Supported
`sb`	FEAT_SB	Armv8-A	Armv8.5-A	Speculation barrier instruction. See Prevention of Speculative execution and data prediction for more information.	Supported
`sha2`	FEAT_SHA256 FEAT_SHA1	Armv8-A	-	Cryptographic extension. See Cryptographic extensions for more information.	Supported

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

For targets in AArch32 state, you can use `-mfpu` to specify the support for floating-point, Advanced SIMD, and Cryptographic extensions.

Table 2-21: AArch32 state M-profile features

Command line `<feature>` option	Optional from	Enabled by default	Description	Support level
`cdecp<N>`	Armv8-M	-	*Custom Datapath Extension* (CDE). `<N>` is in the range 0-7. See Custom Datapath Extension for more information.	Supported
`dsp`	Armv8-M	-	*Digital Signal Processing* (DSP) extension for the Armv8-M.mainline architecture.	Supported
`lob`	Armv8.1-M	Armv8.1-M	Low Overhead Branch extension.	Supported
`mve`	Armv8.1-M	-	*M-Profile Vector Extension* (MVE). See M-profile Vector Extension for more information.	Supported
`pacbti`	Armv8.1-M	-	*Pointer Authentication and Branch Target Identification* (PACBTI) extension. See Armv8.1-M PACBTI extension for more information.	Supported

For architecture extensions supported by Armv8-R AArch32, see Supported architecture features.

Note

To write code that generates instructions for these extensions, use the intrinsics that are described in the Arm C Language Extensions.

Operation

You can use the `-mcpu` option to enable and disable specific architecture features.

To disable a feature, prefix with `no`, for example `cortex-a57+nocrypto`.

To enable or disable multiple features, chain multiple feature modifiers. For example, to enable CRC instructions and disable all other extensions:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57+nocrypto+nofp+nosimd+crc
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If you specify conflicting feature modifiers with `-mcpu`, the rightmost feature is used. For example, the following command enables the Floating-point Extension:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57+nofp+fp
```

You can prevent the use of floating-point instructions or floating-point registers for targets in AArch64 state with the `-mcpu=<name>+nofp+nosimd` option. Subsequent use of floating-point data types in this mode is unsupported.

AArch64 A-profile

`armlink` uses AArch64 library code that can contain floating-point and Advanced SIMD instructions and registers. The use of the AArch64 libraries applies even if you compile the source with `-march=<name>+nofp+nosimd` to prevent the compiler from using floating-point and SIMD instructions and registers. Therefore, there is no guarantee that the linked image for targets in AArch64 state is entirely free of Advanced SIMD and floating-point instructions and registers.

You can prevent the use of Advanced SIMD and floating-point instructions and registers in images that are linked for targets in AArch64 state. Either reimplement the library functions or create your own library that does not use Advanced SIMD and floating-point instructions and registers.

AArch64 R-profile

`armlink` links against AArch64 library code that can use floating-point and SIMD instructions and registers. To use library variants that do not contain floating-point instructions compile with the `+nofp` architecture feature and appropriate linker options. For more information, see How to build for an Armv8-R AArch64 target without hardware floating-point support.

Example: Change the ISA of the cp0 and cp1 encoding space to CDEv1

```
armclang --target=arm-arm-none-eabi -march=armv8.1-m.main+cdecp0+cdecp1 -c test.S
```

Example: List the processors that target the AArch64 state

```
armclang --target=aarch64-arm-none-eabi -mcpu=list
```

Example: Target the AArch64 state of a Cortex-A57 processor

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 test.c
```

Example: Target the AArch32 state of a Cortex-A53 processor, generating A32 instructions

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 -marm test.c
```

Example: Target the AArch32 state of a Cortex-A53 processor, generating T32 instructions

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 -mthumb test.c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Target the AArch32 state of an Arm Neoverse N1 processor

```
armclang --target=arm-arm-none-eabi -mcpu=neoverse-n1 test.c
```

Related information

-marm on page 140
-mthumb on page 193
-mtune=target on page 196
--target on page 221
-mfpu on page 170
--cpu=name (armlink) on page 443
--fpu=name (armlink) on page 470
--coprocN=value (fromelf) on page 776
Volatile variables on page 342
Half-precision floating-point data types on page 342
Half-precision floating-point intrinsics on page 347
Supported architecture feature combinations for specific processors on page 364
Overview of the Realm Management Extension
Custom Datapath Extension support
AArch32 -- Base Instructions (alphabetic order)
A64 -- SIMD and Floating-point Instructions (alphabetic order)
Arm Armv8-A A64 Instruction Set Architecture
Arm Armv9-A A64 Instruction Set Architecture

2.1.77 -mdefault-build-attributes, -mno-default-build-attributes

Adds build attributes to files assembled with the integrated assembler. You can also remove build attributes.


Note

This topic describes a [COMMUNITY] feature. See Support level definitions.

Default

`-mdefault-build-attributes` is the default.

Syntax

```
-mdefault-build-attributes
-mno-default-build-attributes
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Restrictions

Valid only for AArch32 state.

Example: Adding build attributes to assembled files

Create the assembler file `func.s` containing:

```
  // 'func' is a function
  .type func, %function
func:
  bx lr

  // 'value' is a data object:
  .type value, %object
value:
  .word 42
```

Assemble the file with:

```
armclang -c --target=arm-arm-none-eabi -mcpu=cortex-a8 -mdefault-build-attributes -o
 func.o func.s
```

Use `fromelf` to view the build attributes:

```
fromelf --decode_build_attributes func.o

...
** Section #3 '.ARM.attributes' (SHT_ARM_ATTRIBUTES)
    Size   : 43 bytes

    'aeabi' file build attributes:
    0x000000:   05 63 6f 72 74 65 78 2d 61 38 00 06 0a 07 41 08    .cortex-a8....A.
    0x000010:   01 09 02 0a 03 0c 01 22 01 44 01                   .......".D.
        Tag_CPU_name = "cortex-a8"
        Tag_CPU_arch = ARM v7 (=10)
        Tag_CPU_arch_profile = The application profile 'A' (e.g. for Cortex A8)
 (=65)
        Tag_ARM_ISA_use = ARM instructions were permitted to be used (=1)
        Tag_THUMB_ISA_use = Thumb2 instructions were permitted (implies Thumb
 instructions permitted) (=2)
        Tag_FP_arch = VFPv3 instructions were permitted (implies VFPv2 instructions
 were permitted) (=3)
        Tag_Advanced_SIMD_arch = Use of the Advanced SIMD Architecture (Neon) was
 permitted (=1)
        Tag_CPU_unaligned_access = The producer was permitted to generate
 architecture v6-style unaligned data accesses (=1)
        Tag_Virtualization_use = Use of the TrustZone extension was permitted (=1)
...
```

Assemble again with `-mno-default-build-attributes`. The `fromelf` output shows that no build attributes are present.

Related information

Build attributes

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.78 -mexecute-only

Generates execute-only code, and prevents the compiler from generating any data accesses to code sections.

Syntax

```
-mexecute-only
```

Parameters

None.

Restrictions

The following restrictions apply:

- Execute-only code must be T32 code.
- Execute-only code is only supported for:
  - Processors that support the Arm®v8-M architecture, with or without the Main Extension.
  - Processors that support the Armv7-M architecture, such as the Cortex®-M3.
  - Processors that support the Armv6-M architecture. However, execute-only is not supported on Armv6-M for any form of position independent code.
- `-mexecute-only` is incompatible with `-fsanitize=function`. For more information, see -fsanitize, -fno-sanitize.
- 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.
- *Link-Time Optimization* (LTO) does not honor the `armclang` option `-mexecute-only`. If you use the `armclang` options `-flto` or `-Omax`, then the compiler cannot generate execute-only code.

Operation

To keep code and data in separate sections, the compiler disables literal pools and branch tables when using the `-mexecute-only` option.

`-mpure-code` is an alias for `-mexecute-only`.

Related information

Building applications for execute-only memory

2.1.79 -mfloat-abi

Specifies whether to use hardware instructions or software library functions for floating-point operations, and which registers are used to pass floating-point parameters and return values.

Default

The default for `--target=arm-arm-none-eabi` is `softfp`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
-mfloat-abi=<value>
```

Parameters

<value>

One of:

soft

Software library functions for floating-point operations and software floating-point linkage.

softfp

Hardware floating-point instructions and software floating-point linkage.

hard

Hardware floating-point instructions and hardware floating-point linkage.


Note

The `-mfloat-abi` option is not valid with AArch64 targets. AArch64 targets use hardware floating-point instructions and hardware floating-point linkage. However, you can prevent the use of floating-point instructions or floating-point registers for AArch64 targets with the `-mcpu=<name>+nofp+nosimd` option. Subsequent use of floating-point data types in this mode is unsupported.


Note

In AArch32 state, if you specify `-mfloat-abi=soft`, then specifying the `-mfpu` option does not have an effect.

Related information

-mfpu on page 170

2.1.80 -mfpu

Specifies the target *Floating-Point Unit* FPU architecture, that is the floating-point hardware available on the target.

Default

The default FPU option depends on the target processor.

Syntax

To specify a target FPU architecture:

```
-mfpu=<name>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

To view a list of all the supported FPU architectures:

```
-mfpu=list
```

Parameters

<name>

One of the following:

none, softvfp

Use either `-mfpu=none` or `-mfpu=softvfp` to prevent the compiler from using hardware-based floating-point functions. If the compiler encounters floating-point types in the source code, it uses software-based floating-point library functions. This option is similar to the `-mfloat-abi=soft` option.

vfpv3

Enable the Arm®v7 VFPv3 Floating-point Extension. Disable the Advanced SIMD extension.

vfpv3-d16

Enable the Armv7 VFPv3-D16 Floating-point Extension. Disable the Advanced SIMD extension.

vfpv3-fp16

Enable the Armv7 VFPv3 Floating-point Extension, including the optional half-precision extensions. Disable the Advanced SIMD extension.

vfpv3-d16-fp16

Enable the Armv7 VFPv3-D16 Floating-point Extension, including the optional half-precision extensions. Disable the Advanced SIMD extension.

vfpv3xd

Enable the Armv7 VFPv3XD Floating-point Extension, for single precision only. Disable the Advanced SIMD extension.

vfpv3xd-fp16

Enable the Armv7 VFPv3XD Floating-point Extension, for single precision (including the optional half-precision extensions). Disable the Advanced SIMD extension.

neon

Enable the Armv7 VFPv3 Floating-point Extension and the Advanced SIMD extension.

neon-fp16

Enable the Armv7 VFPv3 Floating-point Extension, including the optional half-precision extensions, and the Advanced SIMD extension.

vfpv4

Enable the Armv7 VFPv4 Floating-point Extension. Disable the Advanced SIMD extension.

vfpv4-d16

Enable the Armv7 VFPv4-D16 Floating-point Extension. Disable the Advanced SIMD extension.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

neon-vfpv4
Enable the Armv7 VFPv4 Floating-point Extension and the Advanced SIMD extension.

fpv4-sp-d16
Enable the Armv7 FPv4-SP-D16 Floating-point Extension for single precision only.

fpv5-d16
Enable the Armv7 FPv5-D16 Floating-point Extension. This option disables the scalar half-precision floating-point operations feature. Therefore, because the *M-profile Vector Extension* (MVE) floating-point feature requires the scalar half-precision floating-point operations, this option also disables the MVE floating-point feature, `+mve.fp`. We recommend using the generic `+fp` and `+fp.dp` extension names instead of `-mfpu`. See -mcpu and Supported architecture feature combinations for specific processors for more information.

fpv5-sp-d16
Enable the Armv7 FPv5-SP-D16 Floating-point Extension for single precision only. This option disables the scalar half-precision floating-point operations feature. Therefore, because the MVE floating-point feature requires the scalar half-precision floating-point operations, this option also disables the MVE floating-point feature, `+mve.fp`. We recommend using the generic `+fp` and `+fp.dp` extension names instead of `-mfpu`. See -mcpu and Supported architecture feature combinations for specific processors for more information.

fp-armv8
Enable the Armv8 Floating-point Extension. Disable the Cryptographic Extension and the Advanced SIMD extension.

neon-fp-armv8
Enable the Armv8 Floating-point Extension and the Advanced SIMD extensions. Disable the Cryptographic Extension.

crypto-neon-fp-armv8
Enable the Armv8 Floating-point Extension, the Cryptographic Extension, and the Advanced SIMD extension.

Restrictions

AArch64 state

`-mfpu=list` is not supported when targeting AArch64 state, and the compiler gives an error.

The `-mfpu=<name>` option is ignored with AArch64 targets, that is `aarch64-arm-none-eabi`. Use the `-mcpu` option to override the default FPU for `aarch64-arm-none-eabi` targets. For example, to prevent the use of floating-point instructions or floating-point registers for the `aarch64-arm-none-eabi` target use the `-mcpu=<name>+nofp+nosimd` option. Subsequent use of floating-point data types in this mode is unsupported.

AArch32 state

If you specify `-mfloat-abi=soft` in AArch32 state, then specifying the `-mfpu` option does not have an effect.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

The `-mfpu` option overrides the default FPU option implied by the target architecture.


Note

In Armv7, the Advanced SIMD extension was called the Arm® Neon® Advanced SIMD extension.

The following table shows the floating-point instructions that are available for the Armv8-A and Armv9-A architectures:


Note

For completeness, the table also shows the I8MM instructions for 8-bit integer operations. See Matrix Multiplication extension for more information.

Table 2-22: Floating-point instructions available for FPU architectures

Feature	Armv8.0-A, Armv8.1-A, Armv8.2-A, Armv8.3-A, Armv8.4-A, Armv8.5-A	Armv8.6-A and later	Armv8-R, Armv9.0-A	Armv9.1-A, and later
`-mfpu=fp-armv8`	FP	FP	FP	FP
`-mfpu=neon-fp-armv8`	SIMD, FP	SIMD, FP, BF16, I8MM	SIMD, FP	SIMD, FP, BF16, I8MM
`-mfpu=crypto-neon-fp-armv8`	SIMD, FP	SIMD, FP, BF16, I8MM	SIMD, FP	SIMD, FP, BF16, I8MM

Related information

-mcpu on page 150
-mfloat-abi on page 169
--target on page 221
Supported architecture feature combinations for specific processors on page 364

2.1.81 -mframe-chain

Controls whether a frame chain is maintained, and the level of compliance of the frame records in the frame chain.

Default

The default is `-mframe-chain=none`.

Syntax

```
-mframe-chain=<model>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<model>

The level of compliance of the frame records.

When specifying the level of compliance, it can be one of:

none

Do not maintain compliant frame chain records in the frame chain. The compiler is free to define the contents of the frame chain.

aapcs

Maintain an AAPCS-compliant frame chain record.

aapcs+leaf

Maintain an AAPCS-compliant frame chain record. The `+leaf` modifier means the model is applied to leaf functions in addition to non-leaf functions.



Maintaining AAPCS-compliant frame chain records might have a performance and code size cost. The impact is larger for Thumb® because the frame pointer is a high register that cannot be directly loaded into memory or stored from memory.

For a description of AAPCS-compliant frame chain records see *The Frame Pointer* section in the 32-bit Procedure Call Standard for the Arm Architecture (AAPCS).

Restrictions

This option is only supported on AArch32.

Operation

A frame chain is a linked list of frame activation records that is pointed to directly by the frame pointer.

Use `-mframe-chain` with `-fno-omit-frame-pointer` to enable the frame chain model and store the frame pointer in a register. For more information about `-fno-omit-frame-pointer` see -fomit-frame-pointer, -fno-omit-frame-pointer.

You can use frame chains, for example, to take a snapshot of the call hierarchy quickly and reliably for diagnostic purposes, or to print the call stack when an error is thrown and you cannot easily re-use the custom AArch32 unwinding data.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Option	-fomit-frame-pointer	-fno-omit-frame-pointer
-mframe-chain=none	No frame pointer register is reserved, unless needed for internal compiler usage. No frame chain records are generated.	armclang decides on which frame pointer register to reserve, and might be used by the compiler when required for optimal code generation. Any frame records generated are not guaranteed to be compliant with any frame chain specification. Code external to a function cannot rely on a specific register being used for accessing the frame. Debug information might still describe how the register is used. The compiler might pick a different register for each function. For example, armclang might use R11 for Arm code and R7 for Thumb.
-mframe-chain=aapcs	The AAPCS frame register (r11) is reserved and not used as a general purpose register, but frame records are not required to be generated. If r11 points to an existing frame record, this record is preserved. If code generation requires use of a frame pointer, the frame records that are created are compliant with the AAPCS frame chain specification.	For all functions that modify the link register (r14) an AAPCS-compliant frame record is created on the stack, and the frame pointer (r11) points to this record throughout the function. All non-leaf functions must save the link register.
-mframe-chain=aapcs+leaf	The AAPCS frame register (r11) is reserved and not used as a general purpose register, but frame records are not required to be generated. If r11 points to an existing frame record, this record is preserved. If code generation requires use of a frame pointer, the frame records that are created are compliant with the AAPCS frame chain specification.	An AAPCS-compliant frame record is created for all functions including leaf functions. r11 points to this frame record.

Related information

-fomit-frame-pointer, -fno-omit-frame-pointer on page 77

2.1.82 -mglobal-merge, -mno-global-merge

Enables or disables the merging of global variables when LTO is disabled.

Default

-mglobal-merge is the default at optimization levels, -Ofast and -Oz.

-mno-global-merge is the default at optimization levels, -O0, -O1, -O2, -O3, and -Os.

Syntax

```
-mglobal-merge
-mno-global-merge
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

`-mglobal-merge` overrides `-fdata-sections`.

When global merging combines more than one global variable into a single aggregate, the aggregate must be placed in a single ELF section. If you explicitly place a global variable in its own named section with `__attribute__((section("<name>")))`, `armclang` cannot merge that variable. The compiler still merges global variables that would normally be placed in a separate implicitly named section by `-fdata-sections`.

Related information

-fdata-sections, -fno-data-sections on page 53
-O (armclang) on page 203
__attribute__((section("name"))) variable attribute on page 297

2.1.83 -mharden-pac-ret

Hardens return address signing to protect against a PACMAN brute-force attack that tries to determine valid *Pointer Authentication Codes* (PAC).

Syntax

```
-mharden-pac-ret=<option>
```

Parameters

`<option>`

One of:

`none`

Disables return address signing hardening.

`load-return-address`

Enables return address signing hardening. Can only be used when return address signing is enabled with the `-mbranch-protection=<protection>` command-line option.



This command-line option applies to all functions. However, you can override it on specific functions with the `__attribute__((target("<options>")))` function attribute.

Restrictions

This option is supported on AArch64 targets only.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

Use `-mharden-pac-ret` with `-mbranch-protection=pac-ret` to protect against a PACMAN attack. PACMAN uses speculation to determine PAC, so reduces the protection that pointer authentication provides against *Return Oriented Programming* (ROP) and *Jump Oriented Programming* (JOP) attacks.

The `-mharden-pac-ret` command-line option enables the code generation for the modified return address signing. The compiler uses the `XPACI` instruction that is only available in FEAT_PAUTH. FEAT_PAUTH is mandatory in Arm®v8.3-A and later. For earlier architectures the compiler uses the `XPACLRI` instruction, which only operates on the LR, and is in hint space.

Example: -mharden-pac-ret without the use of attributes

Create the file `address.c` containing the following C code:

```
extern void bar();

int foo(int cond)
{
    bar();
    if (cond)
    {
        return 0;
    }
    return 1;
}
```

Compile `address.c` with the options `-mharden-pac-ret=load-return-address` and `-mbranch-protection=pac-ret+leaf`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.3-a -O2 -mharden-pac-ret=load-
return-address -mbranch-protection=pac-ret+leaf -S -o address.s address.c
```

The compiler generates code with return address signing hardening enabled, and generates an instruction sequence containing the `XPACI` instruction:

```
    ...
foo:
    paciasp
    stp x30, x19, [sp, #-16]!           // 16-byte Folded Spill
    mov w19, w0
    bl  bar
    cmp w19, #0
    cset    w0, eq
    ldp x30, x19, [sp], #16             // 16-byte Folded Reload
    autiasp
    mov x16, x30
    xpaci   x16
    ldr w16, [x16]
    ret
    ...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

To compare against code generated without return address signing hardening enabled, compile `address.c` again without the option `-mharden-pac-ret=load-return-address`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.3-a -O2 -mbranch-
protection=pac-ret+leaf -S -o address.s address.c
```

The generated assembly now contains:

```
    ...
foo:
    paciasp
    stp x30, x19, [sp, #-16]!           // 16-byte Folded Spill
    mov w19, w0
    bl  bar
    cmp w19, #0
    cset    w0, eq
    ldp x30, x19, [sp], #16             // 16-byte Folded Reload
    retaa
    ...
```

The presence of the `RETAA` instruction makes this instruction sequence vulnerable to the PACMAN vulnerability.

`RETAA` is a contraction of the `AUTIASP` and `RET` code sequence. Therefore, this code sequence and `RETAA` are indications that hardening is required. That is, hardening is required if a return address is authenticated, then returned to, with no mitigation code sequence in between.

Example: Return address signing hardening in Armv8.3-A

Create the file `address1.c` containing the following C code:

```
extern void bar();

#ifdef ADD_ATTRIBUTE
__attribute__((target("harden-pac-ret=load-return-address,branch-protection=pac-
ret")))
#endif
int foo(int cond)
{
    bar();
    if (cond)
    {
        return 0;
    }
    return 1;
}
```

Compile `address1.c` with the return address signing option `-mbranch-protection=pac-ret`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.3-a -O2 -mbranch-
protection=pac-ret -S -o address1.s address1.c
```

The generated file `address1.s` contains:

```
    ...
foo:
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    paciasp
    stp x30, x19, [sp, #-16]!           // 16-byte Folded Spill
    mov w19, w0
    bl  bar
    cmp w19, #0
    cset    w0, eq
    ldp x30, x19, [sp], #16             // 16-byte Folded Reload
    retaa
    ...
```

The function `foo` might constitute a PACMAN gadget. An attacker might leverage the speculative execution of the code leading up to the `RETAA` instruction. For more information, see the PACMAN: Attacking ARM Pointer Authentication with Speculative Execution paper.

Compile again without the `-mbranch-protection=pac-ret` option but with the `__attribute__((target("harden-pac-ret=load-return-address,branch-protection=pac-ret")))` function attribute as follows:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.3-a -O2 -DADD_ATTRIBUTE -S -o
 address_attr.s address1.c
```

This command defines the preprocessor macro `ADD_ATTRIBUTE` and adds a function attribute to enable return address signing hardening for the function `foo()`.

The generated assembly now contains:

```
    ...
foo:
    paciasp
    stp x30, x19, [sp, #-16]!           // 16-byte Folded Spill
    mov w19, w0
    bl  bar
    cmp w19, #0
    cset    w0, eq
    ldp x30, x19, [sp], #16             // 16-byte Folded Reload
    autiasp
    mov x16, x30
    xpaci   x16
    ldr w16, [x16]
    ret
    ...
```

Related information

-mbranch-protection on page 144
__attribute__((target("options"))) function attribute on page 276
Security features supported in Arm Compiler for Embedded

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.84 -mharden-sls

Generates code that helps prevent the processor from speculating past an indirect branch that is vulnerable to *Straight-Line Speculation* (SLS).

Syntax

```
-mharden-sls=<option>
```

Parameters

<option>

One of:

all

Enable all mitigations against SLS that are implemented.

blr

Enable the mitigation against SLS for `BLR` instructions.

`armclang` creates a thunk `__llvm_slsblr_thunk_x<N>` for every `X<N>` register. Each thunk is placed in a separate section named `.text.__llvm_slsblr_thunk_x<N>`.

In Arm® Compiler for Embedded 6, the separate thunk code is globally visible and might be called from a location where the `SB` instruction is locally disabled. Therefore, `armclang` always uses the `DSB` and `ISB` speculation barrier instructions after the `BR` instruction.

none

Disable all mitigations against SLS.

retbr

Enable the mitigation against SLS for `RET` and `BR` instructions.

`armclang` uses the `SB` speculation barrier instruction after `RET` and `BR` instructions if that instruction is supported by the target. Otherwise, it uses the `DSB` and `ISB` instructions.

Restrictions

This option is supported on AArch64 targets only.

Example: Mitigation for the BLR instruction

Create the file `function.c` containing the following C code:

```
typedef int functype(int);

int call(functype *funcptr) {
    return funcptr(42) + funcptr(24);
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Compile with the command:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 -S function.c
```

The generated file `function.s` contains:

```
...
    mov w0, #42
    blr x8
    str w0, [sp, #4]                    // 4-byte Folded Spill
    ldr x8, [sp, #8]
    mov w0, #24
    blr x8
    mov w8, w0
...
```

Compile again with the command:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 -mharden-sls=blr -S
 function.c
```

The generated file `function.s` now contains:

```
...
    mov w0, #42
    bl  __llvm_slsblr_thunk_x8
    str w0, [sp, #4]                    // 4-byte Folded Spill
    ldr x8, [sp, #8]
    mov w0, #24
    bl  __llvm_slsblr_thunk_x8
    mov w8, w0
...
    .section
  .text.__llvm_slsblr_thunk_x8,"axG",@progbits,__llvm_slsblr_thunk_x8,comdat
    .hidden __llvm_slsblr_thunk_x8
    .weak   __llvm_slsblr_thunk_x8
    .p2align    4
    .type   __llvm_slsblr_thunk_x8,@function
__llvm_slsblr_thunk_x8:
    mov x16, x8
    br  x16
    dsb sy
    isb
...
```

Example: Mitigation for the RET instruction

Compile the C file from the previous example with the command:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 -S function.c
```

The generated file `function.s` contains:

```
...
    mov w8, w0
    ldr w0, [sp, #4]                    // 4-byte Folded Reload
    add w0, w0, w8
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    ldr x30, [sp, #16]                  // 8-byte Folded Reload
    add sp, sp, #32
    ret
...
```

Compile again with the command:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 -mharden-sls=retbr -S
 function.c
```

The generated file `function.s` now contains:

```
...
    mov w8, w0
    ldr w0, [sp, #4]                    // 4-byte Folded Reload
    add w0, w0, w8
    ldr x30, [sp, #16]                  // 8-byte Folded Reload
    add sp, sp, #32
    ret
    dsb sy
    isb
...
```

Compile again for a target that supports the `SB` instruction:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -mharden-sls=retbr -S
 function.c
```

The generated file `function.s` now has the `SB` instruction after `RET`:

```
...
    mov w8, w0
    ldr w0, [sp, #4]                    // 4-byte Folded Reload
    add w0, w0, w8
    ldr x30, [sp, #16]                  // 8-byte Folded Reload
    add sp, sp, #32
    ret
    sb
...
```

Related information

Straight-line speculation whitepaper
Security features supported in Arm Compiler for Embedded

2.1.85 -mimplicit-it

Specifies the behavior of the integrated assembler if there are conditional instructions outside `IT` blocks.

Default

The default is `-mimplicit-it=arm`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
-mimplicit-it=<name>
```

Parameters

<name>

One of the following:

never

In A32 code, the integrated assembler reports a warning when there is a conditional instruction without an enclosing IT block. In T32 code, the integrated assembler reports an error when there is a conditional instruction without an enclosing IT block.

always

In A32 code, the integrated assembler accepts all conditional instructions without giving an error or warning. In T32 code, the integrated assembler outputs an implicit IT block when there is a conditional instruction without an enclosing IT block. The integrated assembler does not report an error or warning about this.

arm

This is the default. In A32 code, the integrated assembler accepts all conditional instructions without giving an error or warning. In T32 code, the integrated assembler reports an error when there is a conditional instruction without an enclosing IT block.

thumb

In A32 code, the integrated assembler reports a warning when there is a conditional instruction without an enclosing IT block. In T32 code, the integrated assembler outputs an implicit IT block when there is a conditional instruction without an enclosing IT block. The integrated assembler does not give an error or warning about this in T32 code.


Note

This option has no effect for targets in AArch64 state because the A64 instruction set does not include the `IT` instruction. The integrated assembler reports a warning if you use the `-mimplicit-it` option with A64 code.

Related information

IT

2.1.86 -mlittle-endian

Generates code suitable for an Arm processor using little-endian data.

Default

The default is `-mlittle-endian`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
-mlittle-endian
```

Parameters

None.

Related information

-mbig-endian on page 143

2.1.87 -mno-neg-immediates

Disables the substitution of invalid instructions with valid equivalent instructions that use the logical inverse or negative of the specified immediate value.

Default

By default, `armclang` substitutes invalid instructions with an alternative instruction. The substitution must be a valid equivalent instruction that produces the same result by using the logical inverse or negative of the specified immediate value.

Syntax

```
-mno-neg-immediates
```

Parameters

None.

Operation

If an instruction does not have an encoding for the specified value of the immediate operand, but the logical inverse or negative of the immediate operand is available, then `armclang` produces a valid equivalent instruction and inverts or negates the specified immediate value. This applies to both assembly language source files and to inline assembly code in C and C++ language source files.

For example, `armclang` substitutes the instruction `sub r0, r0, #0xFFFFFF01` with the equivalent instruction `add r0, r0, #0xFF`.

You can disable this substitution using the option `-mno-neg-immediates`. In this case, `armclang` generates the error `instruction requires: NegativeImmediates`, if it encounters an invalid instruction that can be substituted using the logical inverse or negative of the immediate value.

When you do not use the option `-mno-neg-immediates`, `armclang` is able to substitute instructions but does not produce a diagnostic message when a substitution has occurred. When you are comparing disassembly listings with source code, be aware that some instructions might have been substituted.

For more information, see A32 and T32 instruction substitutions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

Copy the following code to a file called `neg.s`.

```
    .arm
    sub r0, r0, #0xFFFFFF01
    .thumb
    subw r0, r1, #0xFFFFFF01
```

Assemble the file `neg.s` without the `-mno-neg-immediates` option to produce the output `neg.o`.

```
armclang --target=arm-arm-none-eabi -march=armv7-a -c neg.s -o neg.o
```

Use `fromelf` to see the disassembly from `neg.o`.

```
fromelf --cpu=7-A --text -c neg.o
```

Note that the disassembly from `neg.o` contains substituted instructions `ADD` and `ADDW`:

```
** Section #2 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 8 bytes (alignment 4)
    Address: 0x00000000

    $a.0
          0x00000000:    e28000ff    ....    ADD      r0,r0,#0xff
    $t.1
          0x00000004:    f20100ff    ....    ADDW     r0,r1,#0xff
```

Assemble the file `neg.s` with the `-mno-neg-immediates` option to produce the output `neg.o`.

```
armclang --target=arm-arm-none-eabi -march=armv7-a -c -mno-neg-immediates neg.s -o
 neg.o
```

Note that `armclang` generates the error `instruction requires: NegativeImmediates` when assembling this example with the `-mno-neg-immediates` option.

```
neg.s:2:2: error: instruction requires: NegativeImmediates

 sub r0,#0xFFFFFF01
 ^
neg.s:4:2: error: instruction requires: NegativeImmediates
 subw r0,r1,#0xFFFFFF01
 ^
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.88 -moutline, -mno-outline

These options control an optimization called function outlining, which can reduce code size. This optimization is performed by a component of the compiler known as the outliner.

Default

The default depends on the target and optimization level.

- AArch64 state:

  If the optimization level is `-Oz`, the default is `-moutline`. Otherwise, the default is `-mno-outline`.
- AArch32 state:

  For M-profile targets, the behavior is the same as for AArch64 state.

  For other targets in AArch32 state, the `-Oz` optimization setting does not turn on outlining. The default is `-mno-outline`.

Syntax

```
-moutline
-mno-outline
```

Parameters

None.

Restrictions

Inline assembly might prevent the outlining of functions.

Operation

The outliner searches for identical sequences of code and puts them in a separate function. The outliner then replaces the original sequences of code with calls to this function.

Outlining reduces code size, but it can increase the execution time. The operation of `-moutline` depends on the optimization level and the complexity of the code.


Note

Outlining is not supported on the following targets:

- Arm®v6-M targets.
- Armv8-M targets without the Main Extension.


Tip

The operation of the outliner depends on the complexity of the source code and the size of the resultant object code. The outliner employs heuristics to decide whether outlining a sequence of code can reduce the size of the object code. In some cases, outlining also depends on the instruction set being targeted because of the inherent size differences between T32, A32, and A64 object code. The outliner might identify a code size reduction for A64 code that is not significant for A32 or

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

T32 code. For example, you might see that certain code sequences are outlined when compiling for AArch64 but are not outlined when compiling for AArch32.

You can use the [COMMUNITY] option `-Rpass=machine-outliner` on the `armclang` command line to output feedback from the outliner on how much the code size has been reduced by.

Example: Effect of outlining on a C program when compiling for AArch64 state

Create the file `foo.c` containing the C program:

```
// foo.c

int func1(int x);
int func2(int x);
int func3(int x);

int func1(int x) {
    int i = x;
    i = i * i;
    i += 1;
    i = func3(i);
    i += 2;
    i = func3(i*2);
    return i;
}

int func2(int x) {
    int i = x + 51;
    i = i * i;
    i += 1;
    i = func3(i);
    i += 2;
    i = func3(i*2);
    return i;
}

int func3(int x) {
    return x * x;
}
```

Compile `foo.c` without `-moutline`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a foo.c -S -O2 -o foo.s
```

The compiler-generated assembly with comments removed for brevity is:

```
func1:
    mov w8, #1
    madd    w8, w0, w0, w8
    mul w8, w8, w8
    lsl w8, w8, #1
    add w8, w8, #4
    mul w0, w8, w8
    ret
.Lfunc_end0:

...

func3:
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    mul w0, w0, w0
    ret
.Lfunc_end1:

...

func2:
    add w8, w0, #51
    mov w9, #1
    madd      w8, w8, w8, w9
    mul w8, w8, w8
    lsl w8, w8, #1
    add w8, w8, #4
    mul w0, w8, w8
    ret
.Lfunc_end2:
```

Compile `foo.c` with `-moutline`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -moutline foo.c -S -O2 -o
 foo_outline.s
```

The compiler-generated assembly with comments removed for brevity is:

```
func1:
    mov w8, #1
    madd    w8, w0, w0, w8
    b   OUTLINED_FUNCTION_0
.Lfunc_end0:

...

func3:
    mul w0, w0, w0
    ret
.Lfunc_end1:

...

func2:
    add w8, w0, #51
    mov w9, #1
    madd    w8, w8, w8, w9
    b   OUTLINED_FUNCTION_0
.Lfunc_end2:

...

OUTLINED_FUNCTION_0:
    mul w8, w8, w8
    lsl w8, w8, #1
    add w8, w8, #4
    mul w0, w8, w8
    ret
.Lfunc_end3:
```

An outlined function named `OUTLINED_FUNCTION_0` is created and it contains the instructions that are outlined from `func1()` and `func2()`:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The disassembly is slightly longer with outlining, because of the extra function introduced. However the size of the code component of the binary image is 12 bytes smaller, as reported by the `-Rpass=machine-outliner` option at build time.

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a foo.c -S \
     -O2 -o foo.s -Rpass=machine-outliner

armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -moutline foo.c -S \
     -O2 -o foo_outline.s -Rpass=machine-outliner

remark: Saved 12 bytes by outlining 5 instructions from 2 locations. (Found at:
 foo.c:23:16, foo.c:13:16)
   [-Rpass=machine-outliner]
```

You can use `fromelf` with the `--text -z` command-line option to confirm this:

```
fromelf --text -z foo.o

============================================================================

** Object/Image Component Sizes

 Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

   68          0          0          0          0          0   foo.o
```

```
fromelf --text -z foo_outline.o

============================================================================

** Object/Image Component Sizes

 Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

  56          0          0          0          0          0   foo_outline.o
```

The code component is reduced from 68 bytes to 56 bytes as a result of function outlining.

Related information

-O (armclang) on page 203
-S (armclang) on page 215

2.1.89 -mpixolib

Generates a *Position Independent eXecute Only* (PIXO) library.

Default

`-mpixolib` is disabled by default.

Syntax

```
-mpixolib
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Restrictions


Note

Generation of PIXO libraries is only supported for Arm®v7-M targets.

Generation of PIXO libraries is only supported for C code. However, the application that uses the PIXO library can have C or C++ code.

You cannot generate a PIXO library if your source files contain variadic arguments.

It is not possible for a function in one PIXO library to jump or branch to a symbol in a different PIXO library. Therefore, each PIXO library must contain all the standard library functions it requires. This can result in multiple definitions within the final application.

When linking your application code with your PIXO library:

- The linker must not remove any unused sections from the PIXO library. You can ensure this with the armlink `--keep` command-line option.
- The `RW` sections with `SHT_NOBITS` and `SHT_PROGBITS` must be kept in the same order and same relative offset for each PIXO library in the final image, as they were in the original PIXO libraries before linking the final image.

Operation

Use `-mpixolib` to create a PIXO library, which is a relocatable library containing eXecutable Only code. The compiler ensures that accesses to static data use relative addressing. To access static data in the `RW` section, the compiler uses relative addressing using `R9` as the base register. To access static data in the `RO` section, the compiler uses relative addressing using `R8` as the base registers.

When creating the PIXO library, if you use `armclang` to invoke the linker, then `armclang` automatically passes the linker option `--pixolib` to `armlink`. If you invoke the linker separately, then you must use the `armlink` command-line option `--pixolib`. When creating a PIXO library, you must also provide a scatter file to the linker.

Each PIXO library must contain all the required standard library functions. Arm Compiler for Embedded 6 provides PIXO variants of the standard libraries based on Microlib. You must specify the required libraries on the command-line when creating your PIXO library. These libraries are located in the compiler installation directory under `/lib/pixolib/`.

The PIXO variants of the standard libraries have the naming format `<base>.<endian>`:

`<base>`

`mc_wg`
C library.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

m_wgv
Math library for targets with hardware double precision floating-point support that is compatible with `vfpv5-d16`.

m_wgm
Math library for targets with hardware single precision floating-point support that is compatible with `fpv4-sp-d16`.

m_wgs
Math library for targets without hardware support for floating-point.

mf_wg
Software floating-point library. This library is required when:

- Using `printf()` to print floating-point values.
- Using a math library that does not have all the required floating-point support in hardware. For example if your code has double precision floating-point operations but your target has `fpv4-sp-d16`, then the software floating-point library is used for the double-precision operations.

<endian>

l
Little endian

b
Big endian

Examples

This example shows the command-line invocations for compiling and linking in separate steps, to create a PIXO library from the source file `foo.c`.

```
armclang --target=arm-arm-none-eabi -march=armv7-m -mpixolib -c -o foo.o foo.c

armlink --pixolib --scatter=pixo.scf -o foo-pixo-library.o foo.o mc_wg.l
```

This example shows the command-line invocations for compiling and linking in a single step, to create a PIXO library from the source file `foo.c`.

```
armclang --target=arm-arm-none-eabi -march=armv7-m -mpixolib -Wl,--scatter=pixo.scf
 -o foo-pixo-library.o foo.c mc_wg.l
```

Related information

--pixolib on page 516
--keep=section_id (armlink) on page 482
--startup=symbol, --no_startup on page 539
The Arm C Micro-library

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.90 -mrestrict-it, -mno-restrict-it

Allows or disallows the generation of complex `IT` blocks, and only affects code generated by the compiler.

Default

The default is `-mno-restrict-it`.

Syntax

Disallow generation of complex `IT` blocks:

```
-mrestrict-it
```

Allow generation of complex `IT` blocks:

```
-mno-restrict-it
```

Parameters

None.

Restrictions

This option is supported only for:

- Arm®v8-A targets in T32 state.
- Armv9-A targets in T32 state.

`armclang` does not generate warnings about complex `IT` blocks.

Operation

This option has no effect on controlling warnings about `IT` instructions that were deprecated when a user writes them in assembly language source code or inline assembly. That is, all uses of `IT` that apply to instructions other than a single subsequent 16-bit instruction from a restricted set, and explicit references to the PC within that single 16-bit instruction.

A complex `IT` block consists of an `IT` instruction and any of the following:

- A 32-bit instruction.
- More than a single 16-bit instruction.
- A single 16-bit instruction that is not part of the following set:

Table 2-24: Non-deprecated IT 16-bit conditional instructions

Non-deprecated 16-bit instructions	Class	Notes
`MOV`, `MVN`	Move	Deprecated when `Rm` or `Rd` is the PC.
`LDR`, `LDRB`, `LDRH`, `LDRSB`, `LDRSH`	Load	Deprecated for PC-relative load literal forms

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Non-deprecated 16-bit instructions	Class	Notes
`STR`, `STRB`, `STRH`	Store	-
`ADD`, `ADC`, `RSB`, `SBC`, `SUB`	Add/Subtract	Deprecated for `ADD SP,SP,#imm`, `SUB SP,SP,#imm`, and when `Rm`, `Rdn`, or `Rdm` is the PC
`CMP`, `CMN`	Compare	Deprecated when `Rm` or `Rn` is the PC
`MUL`	Multiply	-
`ASR`, `LSL`, `LSR`, `ROR`	Shift	-
`AND`, `BIC`, `EOR`, `ORR`, `TST`	Logical	-
`BX`, `BLX`	Branch to register	Deprecated when `Rm` is the PC

This option controls whether the compiler is allowed to generate such complex `IT` blocks. The compiler does not report any warnings about assembly language source code that contains such `IT` blocks.

Related information

-mimplicit-it on page 182
IT

2.1.91 -mthumb

Requests that the compiler targets the T32 instruction set.

Default

The default for all targets that support A32 and T32 instructions is `-marm`.

Syntax

```
-mthumb
```

Parameters

None.

Operation

Different architectures support different instruction sets:

- Arm®v8-A and Armv9-A processors in AArch64 state execute A64 instructions.
- Armv8-A and Armv9-A processors in AArch32 state, in addition to Armv7 and earlier A- and R-profile processors execute A32 and T32 instructions.
- M-profile processors execute T32 instructions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

- The `-mthumb` option is not valid for targets in AArch64 state, for example `--target=aarch64-arm-none-eabi`. The compiler ignores the `-mthumb` option and generates a warning when compiling for a target in AArch64 state.
- The `-mthumb` option is recognized when using `armclang` as a compiler, but not when using it as an assembler. To request `armclang` to assemble using the T32 instruction set for your assembly source files, you must use the `.thumb` or `.code 16` directive in the assembly files.


Tip

T32 offers significant code size improvements compared to A32, with comparable performance. Therefore, if you are compiling for AArch32 state for a target that supports both A32 and T32 instructions, consider compiling with `-mthumb` to reduce the size of your code.

Example: Specify the T32 instruction set

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -mthumb test.c
```

Related information

-marm on page 140
--target on page 221
-mcpu on page 150

2.1.92 -mtls-size

Thread Local Storage (TLS) variables are accessed through an offset from a base address. This option controls the maximum size, in address bits, that is used to encode the offset.

Default

The default size is `24`.

We recommend the default value in most cases because TLS variables usually require no more than 16MB.

Syntax

```
-mtls-size=<size>
```

Parameters

`<size>`

One of the following:

- `12`, for an offset up to 4KB.
- `24`, for an offset up to 16MB.
- `32`, for an offset up to 4GB.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- `48`, for an offset up to 256TB. You must also use the `-mcmodel=large` option.

Larger offsets require more instructions to encode the offset.

Restrictions

`-mtls-size` is only supported in AArch64 state.

Related information

-ftls-model on page 105
-mcmodel on page 146
-mtp on page 195

2.1.93 -mtp

Sets the exception level for the *Thread Pointer* (TP) register. The TP registers provide locations to store the IDs of software threads and processes for OS management purposes.

Default

The default value is `tpidr_el0`.



Note

Code that uses the TP register for an exception level can only run at that exception level or higher.

Syntax

```
-mtp=<level>
```

Parameters

`<level>`

One of the following exception levels:

`tpidr_el0` or `el0`

TP register `TPIDR_EL0`.

`tpidr_el1` or `el1`

TP register `TPIDR_EL1`.

`tpidr_el2` or `el2`

TP register `TPIDR_EL2`.

`tpidr_el3` or `el3`

TP register `TPIDR_EL3`.

Restrictions

This option is supported only for AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

The TP register is accessible through one of the following system registers, typically using the `MRS` instruction:

Table 2-25: Thread registers in AArch64 state

Exception level	Thread pointer register
el0	`TPIDR_EL0`
el1	`TPIDR_EL1`
el2	`TPIDR_EL2`
el3	`TPIDR_EL3`

Related information

-ftls-model on page 105
-mtls-size on page 194

2.1.94 -mtune=target

Allows you to tune the code generation for certain scheduling features and optimizations independently from the architecture.



Note

This topic describes a [COMMUNITY] feature. See Support level definitions.



Note

We do not guarantee that your code is optimized as expected when using this option. We recommend that you do not use this option.

Syntax

```
-mtune=<target>
```

Parameters

`<target>`

A processor that supports AArch64 state.

Restrictions

This option is supported only for AArch64 state inputs.

Example: Tune code generation for Cortex-A55

```
armclang -c --target=aarch64-arm-none-eabi -mtune=cortex-a57 -mcpu=cortex-a55 test.c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

-mcpu on page 150

2.1.95 -munaligned-access, -mno-unaligned-access

Enables or disables unaligned accesses to data on Arm processors.

Default

`-munaligned-access` is the default for architectures and processors that support unaligned accesses to data. This default applies to all architectures except for Arm®v6-M and for Armv8-M without the Main Extension.

`-mno-unaligned-access` is the default for architectures and processors that do not support unaligned access to data. This default applies to Armv6-M and to Armv8-M without the Main Extension.

Syntax

```
-munaligned-access
-mno-unaligned-access
```

Parameters

None.

Operation

The compiler defines the `__ARM_FEATURE_UNALIGNED` macro when you specify `-munaligned-access`.

The libraries include special versions of certain library functions designed to exploit unaligned accesses. When unaligned access support is enabled with `-munaligned-access`, the compilation tools use these library functions to take advantage of unaligned accesses. When unaligned access support is disabled with `-mno-unaligned-access`, these special versions are not used.

`-munaligned-access`

Use this option on processors that support unaligned accesses to data, to speed up accesses to packed structures. This option has the following effects:

- When used during compilation, `-munaligned-access` makes `armclang` select sequences of machine instructions that avoid performing unaligned access to memory.
- `-munaligned-access` writes metadata into the output object file. This metadata communicates to `armlink` that `armclang` did not perform unaligned accesses on the object file.


Note

Compiling with this option generates an error for the following architectures:

- Armv6-M.
- Armv8-M without the Main Extension.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

-mno-unaligned-access

If unaligned access is disabled, any unaligned data that is wider than 8 bits is accessed 1 byte at a time. For example, fields wider than 8 bits, in packed data structures, are always accessed 1 byte at a time even if they are aligned.

Related information

Predefined macros on page 336
__attribute__((aligned)) type attribute on page 288
__attribute__((packed)) type attribute on page 289
__attribute__((aligned)) variable attribute on page 292
__attribute__((packed)) variable attribute on page 296
--unaligned_access, --no_unaligned_access (armlink) on page 551
Build attributes
Arm C Language Extensions 2.1
Alignment support in Arm Compiler for Embedded 6

2.1.96 -nobuiltininc

Forces the compiler to exclude the built-in header files.

Default

`-nobuiltininc` is the default, unless `-nostdlibinc` is specified.

Syntax

```
-nobuiltininc
```

Parameters

None.

Operation

Using `-nobuiltininc` means that you can use the `-isystem` option to include your own system header files.

Example: C program used in the -nobuiltininc examples

Create the file `file.c`:

```
#include "math.h"

double foo(double d)
{
    return sqrt(d + 1.0);
}
int main(int argc, char *argv[])
{
    return foo(argc);
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Using the -nostdlib and -nostdlibinc options

Compile with the `-nostdlib` and `-nostdlibinc` options:

```
armclang --target=aarch64-arm-none-eabi -c -nostdlib -nostdlibinc -v file.c
```

`armclang` generates the messages similar to:

```
...
#include "..." search starts here:
#include <...> search starts here:
 <install_directory>\sw\..\sw\ARMCompiler6.21\lib\clang\17\include
End of search list.
```

The built-in include directory is referenced.

Example: Using the -nobuiltininc option

Compile with the `-nobuiltininc` option:

```
armclang --target=aarch64-arm-none-eabi -c -nobuiltininc -v file.c
```

`armclang` generates the following messages:

```
...
#include "..." search starts here:
#include <...> search starts here:
 <install_directory>\sw\..\sw\ARMCompiler6.21\bin\..\include
End of search list.
```

The include directory provided by Arm is referenced.

Example: Using the -nostdlib, -nostdlibinc, and -nobuiltininc options

Compile with the `-nostdlib`, `-nostdlibinc`, and `-nobuiltininc` options:

```
armclang --target=aarch64-arm-none-eabi -c -nostdlib -nostdlibinc -nobuiltininc -v
 file.c
```

`armclang` generates the following messages:

```
...
#include "..." search starts here:
End of search list.
```

The built-in include directory and that provided by Arm are no longer referenced.

Using the combination of `-nobuiltininc` and `-nostdlibinc` is useful if you need to use the Arm FuSa C library. The FuSa C library is different from the Arm standard library because it is designed to work without the built-in header files.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

-isystem <directory> on page 114
-nostdlib on page 200
-nostdlibinc on page 201

2.1.97 -nostdlib

Instructs the compiler not to use the Arm® standard C and C++ libraries.

Default

`-nostdlib` is disabled by default.

Syntax

```
-nostdlib
```

Parameters

None.

Operation

If you use the `-nostdlib` option, `armclang` does not collude with the Arm standard library and only emits calls to functions that the C standard or the AEABI defines. The output from `armclang` works with any ISO C library that is compliant with AEABI.

If you use `armclang` to invoke the linker, `armclang` automatically passes the `--no_scanlib` linker option to `armlink`. If you specify the `armclang` option `-c`, you must invoke `armlink` separately with the `--no_scanlib` option. You must specify the location of the libraries you want to use as input objects to `armlink`, or with the `armlink` option `--userlibpath`.

Note

You must use the `armclang` option `-nostdlib` if:

- You want to use your own libraries instead of the Arm standard libraries.
- You want to reimplement the standard library functions.

  Your libraries must be compliant with the ISO C library and with the AEABI specification.

See also:

- The `-nostdlibinc` option to exclude the Arm standard C and C++ library header files.
- The `-fno-builtin` option to disable special handling of standard C library functions such as `printf()`, `strlen()`, and `malloc()` with inline code or other library functions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Using the -nostdlib option

Create the file `file.c` containing the following C program:

```
#include "math.h"

double foo(double d)
{
    return sqrt(d + 1.0);
}
int main(int argc, char *argv[])
{
    return foo(argc);
}
```

Compiling the file with `-nostdlib` generates a call to `sqrt`, which the C standard defines.

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-
abi=hard -nostdlib

...
      vmov.f64        d17, #1.000000e+00
      vadd.f64        d0, d16, d17
      bl      sqrt
      add     sp, sp, #8
      pop     {r11, pc}
...
```

`armclang` generates a call to `sqrt`, which the C standard defines.

Compile the code again without `-nostdlib`:

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-
abi=hard

...
      vmov.f64        d17, #1.000000e+00
      vadd.f64        d0, d16, d17
      bl      __hardfp_sqrt
      add     sp, sp, #8
      pop     {r11, pc}
...
```

`armclang` generates a call to `__hardfp_sqrt` (from the Arm standard library), which the C standard and the AEABI do not define.

Related information

-nostdlibinc on page 201
-fno-builtin on page 73
Run-time ABI for the Arm Architecture
C Library ABI for the Arm Architecture

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.98 -nostdlibinc

Forces the compiler to exclude the Arm standard C and C++ library header files.

Default

`-nostdlibinc` is disabled by default.

Syntax

```
-nostdlibinc
```

Parameters

None.

Operation

This option still searches the `lib/clang/*/include` directory.

If you want to exclude the built-in header files, then use both the `armclang` options `-nostdlibinc` and `-nobuiltininc`.

If you want to disable the use of the Arm standard library, then use both the `armclang` options `-nostdlibinc` and `-nostdlib`.

Example: Using the -nostdlibinc option

Create the file `file.c`:

```
#include "math.h"

double foo(double d)
{
    return sqrt(d + 1.0);
}
int main(int argc, char *argv[])
{
    return foo(argc);
}
```

Compile this code without `-nostdlibinc`:

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-
abi=hard
```

`armclang` generates a call to `__hardfp_sqrt`, from the Arm standard library:

```
...
    vmov.f64    d17, #1.000000e+00
    vadd.f64    d0, d16, d17
    bl   __hardfp_sqrt
    add sp, sp, #8
    pop {r11, pc}
...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Compile the code again with `-nostdlibinc` and `-nostdlib`:

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-
abi=hard -nostdlibinc -nostdlib
```

`armclang` generates an error because the compiler cannot include the standard library header file `math.h`.

```
file.c:1:10: fatal error: 'math.h' file not found
    1 | #include "math.h"
      |          ^~~~~~~~
```

Related information

-nobuiltininc on page 198
-nostdlib on page 200
-fno-builtin on page 73

2.1.99 -O (armclang)

Specifies the level of optimization to use when compiling source files.

Default

The default is `-O0`. Arm recommends `-O1` rather than `-O0` for the best trade-off between debug view, code size, and performance.

Syntax

```
-O<level>
```

Parameters

`<level>`

One of the following:

`0`

Minimum optimization for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this optimization might result in a larger image.

`1`

Restricted optimization. When debugging is enabled, this option selects a good compromise between image size, performance, and quality of debug view.

Arm recommends `-O1` rather than `-O0` for the best trade-off between debug view, code size, and performance.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2

High optimization. When debugging is enabled, 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 the debug information cannot describe.

3

Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.

fast

Enables the optimizations from both the `armclang` options `-O3` and `-ffp-mode=fast`.


Note

Enabling the aggressive optimizations that the `-ffp-mode=fast` option performs might violate strict compliance with language standards.

max

Maximum optimization. Specifically targets performance optimization. Enables all the optimizations from level `fast`, together with other aggressive optimizations.


Note

This option is not guaranteed to be fully standards-compliant for all code cases.


Note

`-Omax` automatically enables the `armclang` option `-flto`. When `-flto` is enabled, you cannot build ROPI or RWPI images. You can use the generated object files to create static libraries. However, you can link those libraries only with objects created using the same version of `armclang`. This limitation applies to all uses of *Link-Time Optimization* (LTO). Therefore, use of `-Omax` might not be appropriate if you intend to distribute static libraries, unless potential users are aware of this restriction.


Note

When using `-Omax`:

- Code-size, build-time, and the debug view can each be adversely affected.
- Arm cannot guarantee that the best performance optimization is achieved in all code cases.
- Meaningful disassembly cannot be output when the `-flto` option is enabled. The reason is because the `-flto` option is turned on by default at `-Omax`, and that option generates files containing bitcode.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- If you are trying to compile at `-Omax` and have separate compile and link steps, then also include `-Omax` on your `armlink` command line.


Note

LTO does not honor the `armclang -mexecute-only` option. If you use the `armclang -flto` or `-Omax` options, then the compiler cannot generate execute-only code.

s

Performs optimizations to reduce code size, balancing code size against code speed.

z

Performs optimizations to minimize image size.


Note

This option generates literal pools instead of a `MOVW`/`MOVT` pair of instructions for improved code size.

min

Specifically targets reducing code size. Enables all the optimizations from level `-Oz`, together with:

- A basic set of LTO aimed at removing unused code and data, while also trying to optimize global memory accesses.
- Virtual function elimination. This is a particular benefit to C++ users.


Note

`-Omin` automatically enables the `armclang` option `-flto`. When `-flto` is enabled, you cannot build ROPI or RWPI images. The generated object files can be used for creating static libraries, but those libraries can only be linked with objects also created using the same version of `armclang`. This limitation applies to all uses of LTO. Therefore, use of `-Omin` might not be appropriate if you intend to distribute static libraries, unless potential users are aware of this restriction.


Note

When using `-Omin`:

- Performance, build-time, and the debug view can each be adversely affected.
- Arm cannot guarantee that the best code size optimization is achieved in all code cases.
- Meaningful disassembly cannot be output when the `-flto` option is enabled. The reason is because the `-flto` option is turned on by default at `-Omin`, and that option generates files containing bitcode.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- If you are trying to compile at `-Omin` and have separate compile and link steps, then also include `-Omin` on your `armlink` command line.

Operation

The optimization level is not a simple scale. Some optimization levels improve performance, possibly at the expense of larger output code size. Other optimization levels target reduced code size instead of performance. Some levels are intended to provide a balance between the two goals. Decide what your optimization goals are and choose a level that best suits your requirements.

Optimization goal	Useful optimization levels
Smaller code size	`-Oz`, `-Omin`
Faster performance	`-O2`, `-O3`, `-Ofast`, `-Omax`
Good debug experience without code bloat	`-O1`
Better correlation between source code and generated code	`-O0` (no optimization)
Faster compile and build time	`-O0` (no optimization)
Balanced code size reduction and fast performance	`-Os`


Note

- The optimization level to use for the best code coverage might depend on your source code.
- *Scalable Vector Extension* (SVE) auto-vectorization is enabled by default at the `-O2` and `-O3` levels. Auto-vectorization is identical at both levels, however `-O3` results in higher general code optimization. You can also use the `fvectorize` option to turn on auto-vectorization with the `-O1` optimization level, or `fno-vectorize` to turn auto-vectorization off with higher optimization levels.

Related information

-flto, -fno-lto on page 72
-ffp-mode on page 62
-fropi, -fno-ropi on page 80
-frwpi, -fno-rwpi on page 82
-fvectorize, -fno-vectorize on page 109
Selecting optimization options
Optimizing for code size or performance
Restrictions with Link-Time Optimization
Literal pool options in armclang

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.100 -o (armclang)

Specifies the name of the output file.

Default

If you do not specify a `-o` option, the compiler names the output file according to the conventions described in the following table:

Table 2-27: Compiling without the -o option

Compiler option	Action	Operation notes
`-c`	Produces an object file whose name defaults to `filename.o` in the current directory. `<filename>` is the name of the source file stripped of any leading directory names.	-
`-S`	Produces an assembly file whose name defaults to `filename.s` in the current directory. `<filename>` is the name of the source file stripped of any leading directory names.	-
`-E`	Writes output from the preprocessor to the standard output stream	-
(No option)	Produces temporary object files, then automatically calls the linker to produce an executable image with the default name of `a.out`	None of `-o`, `-c`, `-E` or `-S` is specified on the command line

Syntax

```
-o <filename>
```

Parameters

`<filename>`

The name of the output file.

Operation

The option `-o <filename>` specifies the name of the output file produced by the compiler.

The option `-o-` redirects output to the standard output stream when used with the `-c` or `-S` options.

2.1.101 -pedantic

Generate warnings if code violates strict ISO C and ISO C++.

Default

`-pedantic` is disabled by default.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
-pedantic
```

Parameters

None.

Operation

If you use the `-pedantic` option, the compiler generates warnings if your code uses any language feature that conflicts with strict ISO C or ISO C++.



Note

The `-pedantic` option is stricter than the `-Wpedantic` option.

Example

```
void foo(void)
{
    long long i; /* okay in nonstrict C90 */
}
```

Compiling this code with `-pedantic` generates a warning.

```
armclang --target=arm-arm-none-eabi -march=armv8-a file.c -c -std=c90 -pedantic
```

2.1.102 -pedantic-errors

Generate errors if code violates strict ISO C and ISO C++.

Default

`-pedantic-errors` is disabled by default.

Syntax

```
-pedantic-errors
```

Parameters

None.

Operation

If you use the `-pedantic-errors` option, the compiler does not use any language feature that conflicts with strict ISO C or ISO C++. The compiler generates an error if your code violates strict ISO language standard.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

```
void foo(void)
{
    long long i; /* okay in nonstrict C90 */
}
```

Compiling this code with `-pedantic-errors` generates an error:

```
armclang --target=arm-arm-none-eabi -march=armv8-a file.c -c -std=c90 \
    -pedantic-errors
```

2.1.103 --print-enabled-extensions

Print the architecture extensions enabled by the target options specified to the compiler for AArch64 state.

Use this option to check if the target options specified to the compiler result in the required architecture features being enabled correctly for your project.

Syntax

```
--print-enabled-extensions
```

Parameters

None.

Restrictions

Supported in AArch64 state only.

Operation

The compiler enables different architecture extensions depending on the target options specified to the compiler. The target options include:

- The `-march=<name>` option.
- The `-mcpu=<name>` option.
- `+<feature>` or `+no<feature>` feature modifiers specified with a `-march=<name>` or `-march=<name>` option.

The `--print-enabled-extensions` option prints the set of architecture extensions that are enabled by these target options.

Additionally, this option disables the generation of an output file.

The output of `--print-enabled-extensions` consists of the following items for each enabled architecture extension:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The feature identifier, of the form `FEAT_<name>`. For more information about feature identifiers, see Feature names in the A-profile architecture.
- A description of the enabled architecture extension.

Example: Check the default extensions enabled for Armv9.2-A

Run the following command:

```
armclang --target=aarch64-arm-none-eabi -march=armv9.2-a --print-enabled-extensions
```

The `--print-enabled-extensions` option causes the compiler to print the following list of enabled extensions, with the additional spacing removed for clarity:

```
Extensions enabled for the given AArch64 target

     Architecture Feature(s)     Description
     FEAT_AMUv1                  Enable Armv8.4-A Activity Monitors extension
     FEAT_AMUv1p1                Enable Armv8.6-A Activity Monitors Virtualization
 support
     FEAT_AdvSIMD                Enable Advanced SIMD instructions
     FEAT_BF16                   Enable BFloat16 Extension
     FEAT_BTI                    Enable Branch Target Identification
     FEAT_CCIDX                  Enable Armv8.3-A Extend of the CCSIDR number of
 sets
     FEAT_CRC32                  Enable Armv8.0-A CRC-32 checksum instructions
     FEAT_CSV2_2                 Enable architectural speculation restriction
     FEAT_DIT                    Enable Armv8.4-A Data Independent Timing
 instructions
     FEAT_DPB                    Enable Armv8.2-A data Cache Clean to Point of
 Persistence
     FEAT_DPB2                   Enable Armv8.5-A Cache Clean to Point of Deep
 Persistence
     FEAT_DotProd                Enable dot product support
     FEAT_ECV                    Enable enhanced counter virtualization extension
     FEAT_ETE                    Enable Embedded Trace Extension
     FEAT_FCMA                   Enable Armv8.3-A Floating-point complex number
 support
     FEAT_FGT                    Enable fine grained virtualization traps extension
     FEAT_FP                     Enable Armv8.0-A Floating Point Extensions
     FEAT_FP16                   Enable half-precision floating-point data
 processing
     FEAT_FRINTTS                Enable FRInt[32|64][Z|X] instructions that round a
 floating-point number to an integer (in FP format) forcing it to fit into a 32- or
 64-bit int
     FEAT_FlagM                  Enable Armv8.4-A Flag Manipulation instructions
     FEAT_FlagM2                 Enable alternative NZCV format for floating point
 comparisons
     FEAT_HCX                    Enable Armv8.7-A HCRX_EL2 system register
     FEAT_I8MM                   Enable Matrix Multiply Int8 Extension
     FEAT_JSCVT                  Enable Armv8.3-A JavaScript FP conversion
 instructions
     FEAT_LOR                    Enable Armv8.1-A Limited Ordering Regions
 extension
     FEAT_LRCPC                  Enable support for RCPC extension
     FEAT_LRCPC2                 Enable Armv8.4-A RCPC instructions with Immediate
 Offsets
     FEAT_LSE                    Enable Armv8.1-A Large System Extension (LSE)
 atomic instructions
     FEAT_LSE2                   Enable Armv8.4-A Large System Extension 2 (LSE2)
 atomicity rules
     FEAT_MEC                    Enable Memory Encryption Contexts Extension
     FEAT_MPAM                   Enable Armv8.4-A Memory system Partitioning and
 Monitoring extension
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
      FEAT_NV, FEAT_NV2           Enable Armv8.4-A Nested Virtualization
 Enchancement
      FEAT_PAN                    Enable Armv8.1-A Privileged Access-Never extension
      FEAT_PAN2                   Enable Armv8.2-A PAN s1e1R and s1e1W Variants
      FEAT_PAuth                  Enable Armv8.3-A Pointer Authentication extension
      FEAT_RAS, FEAT_RASv1p1      Enable Armv8.0-A Reliability, Availability and
 Serviceability Extensions
      FEAT_RDM                    Enable Armv8.1-A Rounding Double Multiply Add/
Subtract instructions
      FEAT_RME                    Enable Realm Management Extension
      FEAT_SB                     Enable Armv8.5-A Speculation Barrier
      FEAT_SEL2                   Enable Armv8.4-A Secure Exception Level 2
 extension
      FEAT_SPECRES                Enable Armv8.5-A execution and data prediction
 invalidation instructions
      FEAT_SSBS, FEAT_SSBS2       Enable Speculative Store Bypass Safe bit
      FEAT_SVE                    Enable Scalable Vector Extension (SVE)
 instructions
      FEAT_SVE2                   Enable Scalable Vector Extension 2 (SVE2)
 instructions
      FEAT_TLBIOS, FEAT_TLBIRANGE Enable Armv8.4-A TLB Range and Maintenance
 instructions
      FEAT_TRBE                   Enable Trace Buffer Extension
      FEAT_TRF                    Enable Armv8.4-A Trace extension
      FEAT_UAO                    Enable Armv8.2-A UAO PState
      FEAT_VHE                    Enable Armv8.1-A Virtual Host extension
      FEAT_WFxT                   Enable Armv8.7-A WFET and WFIT instruction
      FEAT_XS                     Enable Armv8.7-A limited-TLB-maintenance
 instruction
```

Example: Check which extensions are enabled for Armv9.2-A targets with the +fp8dot4 feature modifier

The `+fp8dot4` feature modifier can be used to enable the Advanced SIMD, SVE2, and SME2 FP8 instructions using the E5M2 and E4M3 8-bit floating-point formats feature, FEAT_FP8DOT4.

FEAT_FP8DOT4 depends on multiple other architecture extensions.

To check the architecture extensions enabled with the `+fp8dot4` feature modifier for an Arm®v9.2-A target, run the following command and compare its output with the output shown in the previous example:

```
armclang --target=aarch64-arm-none-eabi -march=armv9.2-a+fp8dot4 --print-enabled-
extensions
```

The additional extensions compared to the output shown in the previous example correspond to architecture extensions enabled for Armv9.2-A targets with the `+fp8dot4` feature modifier:

```
Extensions enabled for the given AArch64 target

      Architecture Feature(s)     Description
      ...
      FEAT_FAMINMAX               Enable FAMIN and FAMAX instructions
      FEAT_FP8                    Enable FP8 instructions
      FEAT_FP8DOT4                Enable FP8 4-way dot instructions
      FEAT_FP8FMA                 Enable Armv9.5-A FP8 multiply-add instructions
      FEAT_LUT                    Enable Lookup Table instructions
```

The additional spacing is removed for clarity.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Check the differences in extensions between Armv9.5-A and Armv9.6-A

`--print-enabled-extensions` is useful to check the differences in the extensions supported by different Arm architectures. This example shows the differences between Armv9.5-A and Armv9.6-A:

```
armclang --target=aarch64-arm-arm-eabi -march=armv9.5-a --print-enabled-extensions

armclang --target=aarch64-arm-arm-eabi -march=armv9.6-a --print-enabled-extensions
```

Comparing the outputs shows that the following additional extensions are available in Armv9.6-A:

```
Extensions enabled for the given AArch64 target

      Architecture Feature(s)     Description
      ...
      FEAT_CMPBR     Enable A64 base compare and branch instructions
      ...
      FEAT_FPRCVT    Enable A64 base convert instructions with only SIMD&FP scalar
 register operands
      ...
      FEAT_LSUI      Enable Armv9.6-A unprivileged load/store instructions
      ...
      FEAT_OCCMO     Enable Armv9.6-A Outer cacheable cache maintenance operations
      ...
      FEAT_SME       Enable Scalable Matrix Extension (SME)
      FEAT_SME2      Enable Scalable Matrix Extension 2 (SME2) instructions
      FEAT_SME2p1    Enable Scalable Matrix Extension 2.1 instructions
      FEAT_SME2p2    Enable Scalable Matrix Extension 2.2 instructions
      ...
      FEAT_SVE2p2    Enable Armv9.6-A Scalable Vector Extension 2.2 instructions
      ...
```

The common extensions and additional spacing are removed for clarity.

Related information

-march on page 123
-mcpu on page 150

2.1.104 -resource-dir

Identifies the location of resource files that are used by various `armclang` features.

Syntax

```
-resource-dir=<path_to_resource_folder>
```

Parameters

`<path_to_resource_folder>`

Location of the resource folder containing resource files required by various `armclang` features.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

`-fsanitize=cfi` expects the ignore list file to be in a `share` folder under `<path_to_resource_folder>`.

Examples

To specify the location of the `/work/project/resources/share/cfi_ignorelist.txt` file when using the *Control Flow Integrity* (CFI) sanitizer scheme checks, specify:

```
-fsanitize=cfi -resource-dir=/work/project/resources
```

Related information

-fsanitize, -fno-sanitize on page 84

2.1.105 -Rpass

Outputs remarks from the optimization passes made by `armclang`. You can output remarks for all optimizations, or remarks for a specific optimization.

Note

This topic describes a [COMMUNITY] feature. See Support level definitions.

Syntax

```
-Rpass={.*|<optimization>}
-Rpass-missed={.*|<optimization>} [COMMUNITY]
```

Parameters

`.*`

Indicates that remarks for all major optimizations such as inlining, vectorization, and loop optimizations are to be reported. However, not all optimization passes support this feature.

`<optimization>`

Is a specific optimization for which remarks are to be output. See the Clang Compiler User's Manual for more information about the optimization values you can specify.

Example: C file used in the inlining and stack protection examples

Create the file `test.c` containing the following C code:

```
// test.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
void *__stack_chk_guard = (void *)0xdeadbeef;

void __stack_chk_fail(void) {
  printf("Stack smashing detected.\n");
  exit(1);
}

static void copy(const char *p) {
  char buf[9];
  strcpy(buf, p);
  printf("Copied: %s\n", buf);
}

int main(void) {
  const char *t = "Hello World!";
  copy(t);
  printf("%s\n", t);
  return 0;
}
```

Example: Display the inlining remarks

Compile `test.c` with:

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -O2 -Rpass=inline test.c

test.c:21:3: remark: 'copy' inlined into 'main' with (cost=-14985, threshold=337) at
 callsite main:2:3; [-Rpass=inline]
   21 |   copy(t);
      |   ^
```

Example: Display the stack protection remarks

Compile `test.c` with:

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -O0 -fstack-protector -
Rpass=stack-protector test.c

test.c:13:13: remark: Stack protection applied to function copy due to a stack
 allocated buffer or struct containing a
      buffer [-Rpass=stack-protector]
   13 | static void copy(const char *p) {
      |             ^
```

Example: Display functions marked with always_inline that are not inlined

Create the file `recurse.c` containing the following C code:

```
// recurse.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int multiply_by_nine(int a) __attribute__((always_inline));
int add_digits(int b) __attribute__((always_inline));

const int NINE = 9;

int multiply_by_nine(int a) {
    return a * NINE;
}

// Recursive function to add the digits in a number
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
// until a single digit remains
int add_digits(int b) {
    int digit;
    int total = 0;
    int length = snprintf(NULL, 0, "%d", b);
    // Return 0 if number too big or value is 0
    if ((length > 10) || (b == 0)) {
        return 0;
    }
    char* number = malloc(length + 1);
    snprintf(number, length + 1, "%d", b);
    // Number has a single digit, so return the value
    if (length == 1) {
        free(number);
        return b;
    }
    // Add the individual digits in the current number
    for (int i = 0; i < length; i++) {
        digit = number[i] - '0';
        total = total + digit;
    }
    // Number still has more than 1 digit, so continue
    // to add the digits
    if (total > NINE) {
        free(number);
        total = add_digits(total);
    }
    free(number);
    return total;
}

int main(void) {
    int a = 789;
    int b = multiply_by_nine(a);
    int c = add_digits(b);
    if (c != 0) {
        printf("Multiplying %d by %d, then adding all the digits of the result is:
 %d\n", NINE, a, c);
    }
    return 0;
}
```

Compile `recurse.c` with:

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -O2 -Rpass-missed=inline
 recurse.c

recurse.c:41:17: remark: 'add_digits' not inlined into 'add_digits' because it
 should never be inlined (cost=never):
      recursive call [-Rpass-missed=inline]
   41 |         total = add_digits(total);
      |                 ^
recurse.c:50:13: remark: 'add_digits' not inlined into 'main' because it should
 never be inlined (cost=never): recursive
      call [-Rpass-missed=inline]
   50 |     int c = add_digits(b);
      |             ^
```

Related information

-fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector on page 100

__attribute__((always_inline)) function attribute on page 259

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.106 -S (armclang)

Outputs the disassembly of the machine code that the compiler generates.

Syntax

```
-S
```

Parameters

None.

Operation

Object modules are not generated. The name of the assembly output file defaults to `<filename>.s` in the current directory. `<filename>` is the name of the source file with any leading directory names removed. The default filename can be overridden with the `-o` option.


Note

It is not possible to output meaningful disassembly when the `-flto` option is enabled because this option generates files containing bitcode. The `-flto` option is enabled by default at `-Omax`.

Related information

-o (armclang) on page 206
-O (armclang) on page 203
-flto, -fno-lto on page 72

2.1.107 -save-temps

Instructs the compiler to generate intermediate files in various formats from the specified C or C++ file.

Syntax

```
-save-temps
```

Parameters

None.

Restrictions

Use only with the `-c` or `-S` options.

Operation

This option outputs the following files, that are listed in the order they are created:

- `<filename>.i` for C or `<filename>.ii` for C++. That is, the C or C++ file after pre-processing.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- `<filename>.bc`: the llvm-ir bitcode file.
- `<filename>.s`: the assembly file. The intermediate assembly file content is similar to the output from disassembling object code that is compiled from C or C++.
- `<filename>.o`: the output object file.

`<filename>` is the name of the input C or C++ file. If you specify multiple C or C++ files, then the compiler creates intermediate files for each input file.


Note

- Specifying `-c` means that the compilation process stops after the compilation step, and does not do any linking.

  When you use `-save-temps` with `-c`, the compiler produces the `.s` assembly then invokes the assembler on the `.s` file to produce the `.o` object file.
- Specifying `-S` means that the compilation process stops after the disassembly step, and does not create an object file.

Example: hello.c used in the -save-temps examples

Create the following `hello.c` file:

```
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");
}
```

Example: Use -save-temps with -c and -v

The following command outputs intermediate files with the filename `hello`, including the `hello.o` file. Using `-v` shows the command invocations:

```
armclang --target=aarch64-arm-none-eabi -save-temps -c -v hello.c
```

Compare the output from this command with the output from the following command:

```
armclang --target=aarch64-arm-none-eabi -c -v hello.c
```

This comparison shows an extra command invocation to assemble the intermediate `hello.s` file to an object file.

Example: Use -save-temps with -S and -v

The following command outputs intermediate files with the filename `hello`, but does not output a `hello.o` file. Using `-v` shows the command invocations:

```
armclang --target=aarch64-arm-none-eabi -save-temps -S -v hello.c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

-c (armclang) on page 43
-S (armclang) on page 215

2.1.108 -shared (armclang)

Creates a *System V* (SysV) shared object.

Default

This option is disabled by default.

Syntax

```
-shared
```

Parameters

None.

Operation

This option causes the compiler to invoke `armlink` with the `--shared` option when performing the link step.

You must use this option with `-fsysv` and `-fpic`.

2.1.109 -std

Specifies the language standard to compile for.


Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.


Note

Arm does not guarantee the compatibility of C++ compilation units compiled with different major or minor versions of Arm® Compiler for Embedded and linked into a single image. Therefore, Arm recommends that you always build your C++ code from source with a single version of the toolchain. You can mix C++ with C code or C libraries.

Default

For C++ code, the default is `gnu++17`. For more information about C++ support, see *C++ Status* at https://clang.llvm.org.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

For C code, the default is `gnu11`. For more information about C support, see *Language Compatibility* at https://clang.llvm.org.

Syntax

```
-std=<name>
```

Parameters

<name>

Specifies the language mode. Valid values include:

c90

C as defined by the 1990 C standard.

gnu90

C as defined by the 1990 C standard, with extra GNU extensions.

c99

C as defined by the 1999 C standard.



Arm Compiler for Embedded does not conform to the detailed specification for IEC 60559 compatibility in complex number arithmetic that is described in C99 Annex G. Therefore, the feature macro `__STDC_IEC_559_COMPLEX__` is not defined.

gnu99

C as defined by the 1999 C standard, with extra GNU extensions.

c11 [COMMUNITY]

C as defined by the 2011 C standard.

gnu11 [COMMUNITY]

C as defined by the 2011 C standard, with extra GNU extensions.

c++98

C++ as defined by the 1998 C++ standard.

gnu++98

C++ as defined by the 1998 C++ standard, with extra GNU extensions.

c++03

C++ as defined by the 2003 C++ standard.

gnu++03

C++ as defined by the 2003 C++ standard, with extra GNU extensions.

c++11

C++ as defined by the 2011 C++ standard.

gnu++11

C++ as defined by the 2011 C++ standard, with extra GNU extensions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

c++14
C++ as defined by the 2014 C++ standard.

gnu++14
C++ as defined by the 2014 C++ standard, with extra GNU extensions.

c++17
C++ as defined by the 2017 C++ standard.

gnu++17
C++ as defined by the 2017 C++ standard, with extra GNU extensions.

For C and C++ language support and libc++ library support in Arm Compiler for Embedded 6, see:

- C language
- C++ language
- libc++ C++14
- libc++ C++17


Note

Use of `c11` library features is unsupported.


Note

Any other C and C++ language standards that are accepted by the `-std` option are [COMMUNITY] features. Library features associated with such standards are unsupported and likely to be incomplete.


Note

Some GNU extensions that are compatible with the C or C++ language standard might be enabled when you specify a `c` or `c++` language mode.


Note

`armclang` always applies the rules for type auto-deduction for copy-list-initialization and direct-list-initialization from C++17, regardless of which C++ source language mode a program is compiled for. For example, the compiler always deduces the type of `foo` as `int` instead of `std::initializer_list<int>` in the following code:

```
auto foo{ 1 };
```

Related information

-faligned-new on page 49

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Standard C Implementation Definition on page 1044
Standard C++ Implementation Definition on page 1065
Support level definitions on page 30
Language Compatibility in LLVM Clang
C++ implementation status in LLVM Clang
Tools and libraries provided with Arm Compiler for Embedded 6

2.1.110 --target

Generate code for the specified target triple.

Default

The `--target` option is mandatory and has no default. You must always specify a target triple.

Syntax

```
--target=<triple>
```

Parameters

`<triple>`

The target triple has the form `<architecture>-<vendor>-<OS>-<abi>`. Supported target triples are:

`aarch64-arm-none-eabi`

Generates A64 instructions for AArch64 state. Implies `-march=armv8-a` unless `-mcpu` or `-march` is specified.

`arm-arm-none-eabi`

Generates A32/T32 instructions for AArch32 state. Must be used in conjunction with `-march` (to target an architecture) or `-mcpu` (to target a processor).


Note

- The target triples are case-sensitive.
- The `--target` option is an `armclang` option. For all of the other tools, such as `armasm` and `armlink`, use the `--cpu` and `--fpu` options to specify target processors and architectures.
- *Scalable Vector Extension* (SVE) is an extension to AArch64 state. Therefore, the only supported target for this release is `aarch64-arm-none-eabi`.

Related information

-marm on page 140
-mthumb on page 193
-mcpu on page 150
-mfpu on page 170

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.111 -U

Removes any initial definition of the specified macro.

Syntax

```
-U <name>
```

Parameters

<name>

The name of the macro to be undefined.

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.

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>`.

Operation

Specifying `-U<name>` has the same effect as placing the text `#undef <name>` at the head of each source file.

Related information

-D (armclang) on page 44
Predefined macros on page 336
-include on page 114

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.112 -u (armclang)

Prevents the removal of a specified symbol if it is undefined.

Syntax

```
-u <symbol>
```

Parameters

<symbol>

The symbol to keep.

Operation

If you use `armclang` to invoke the linker, `armclang` automatically translates this option to `--undefined` and passes it to `armlink`. If you specify the `armclang` option `-c`, you must invoke `armlink` separately with the `--undefined` option.

See --undefined=symbol for information about the `--undefined` linker option.

Related information

--undefined=symbol on page 551

2.1.113 -v (armclang)

Displays the commands that invoke the compiler and sub-tools, such as `armlink`, and executes those commands.

Syntax

```
-v
```

Parameters

None.

Operation

The `-v` compiler option produces diagnostic output showing exactly how the compiler and linker are invoked, displaying the options for each tool. The option also displays the include directories and the version information.

With the `-v` option, `armclang` displays this diagnostic output and executes the commands.



Note

To display the diagnostic output without executing the commands, use the `-###` option.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information
--verbose (armclang) on page 224
-### on page 230

2.1.114 --verbose (armclang)

Displays the same information as `-v`.

Syntax
```
--verbose
```

Parameters
None.

Related information
-v (armclang) on page 223
-### on page 230

2.1.115 --version (armclang)

Displays the same information as `--vsn`.

Syntax
```
--version
```

Parameters
None.

Related information
--vsn (armclang) on page 225

2.1.116 --version_number (armclang)

Displays the version of armclang that you are using.

Syntax
```
--verbose
```

Parameters
None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

`armclang` displays the version number in the format `<M><mm><uu><xx>`, where:

- `<M>` is the major version number, `6`.
- `<mm>` is the minor version number.
- `<uu>` is the update number.
- `<xx>` is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions.

Related information

Predefined macros on page 336

2.1.117 --vsn (armclang)

Displays the version information and the license details.

Syntax

```
--vsn
```

Parameters

None.

Operation

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm® Compiler for Embedded you are using. If you need to access the version in other tools or scripts, for example in build scripts, use the output from `--version_number`.

Table 2-28: Arm Compiler for Embedded Version Information

Field	Description
`Product: <Product Name>`	The product name, which depends on the currently activated license or suite that contains Arm Compiler for Embedded as a component, such as: • Arm *Software Success Kit* (SSK) with an SSK license entitlement. • Arm Development Studio. • Arm® Keil® *Microprocessor Development Kit* (MDK). For example, in Arm Development Studio, `--vsn` lists the product as: `Arm Development Studio <edition> Edition <version>`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Field	Description
`Component: <component>`	The version of Arm Compiler for Embedded. The version is made up of the major, minor, and update number: `Arm Compiler for Embedded <Major Version>.<Minor Version>.<Update Version>`
`Tool: <tool name> [<tool identifier>]`	The tool name and tool identifier. The tool identifier must not be used to identify versions of Arm Compiler for Embedded. The tool identifier is different for each operating system.
`Target name: <target_name>`	The value of the target triple specified with the `--target` option.

Example: armclang --vsn output for Arm Development Studio

The `armclang` output for Arm Compiler for Embedded 6.23 on Linux x86 with Arm Development Studio Gold Edition 2024.0:

```
Product: Arm Development Studio Gold Edition 2024.0
Component: Arm Compiler for Embedded 6.23
Tool: armclang [<toolid>]

Target: <target_name>
```

Related information

--version (armclang) on page 224
--version_number (armclang) on page 224

2.1.118 -W (armclang)

Controls the diagnostics that `armclang` reports.

Syntax

```
-W<name>
```

Parameters

<name>

Table 2-29: General warnings and errors

<name>	Description
`all`	Enables all the warnings about questionable constructs and some language-specific warnings. This option does not enable some warnings that are difficult to avoid and there is no simple way of suppressing those warnings.
`everything`	Enable all possible warnings, and includes warnings that are often not helpful. We recommend that you do not use this option.
`c99-extensions`	Warns about any use of C99-specific features.
`c11-extensions`	Warns about any use of C11-specific features.
`c2x-extensions`	Warns about any use of C2x-specific features.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<name>	Description
c++11-extensions	Warns about any use of C++11-specific features.
c++14-extensions	Warns about any use of C++14-specific features.
c++17-extensions	Warns about any use of C++17-specific features.
c++20-extensions	Warns about any use of C++20-specific features.
c++2b-extensions	Warns about any use of C++2b-specific features.
error	Turn warnings into errors.
gnu	For warnings about GCC extensions.
microsoft	For warnings about Microsoft (MSVC) extensions.
license-management	For license management warnings.
pedantic	Generate warnings if code violates strict ISO C and ISO C++.
unsupported-option	For warnings about options that are not supported.
format=<n>	Checks calls to functions such as `printf` and `scanf`. This option ensures that the supplied arguments match the types specified in the format string. For more information about `-Wformat`, see Options to Request or Suppress Warnings.

Table 2-30: Controlling warnings and errors for specific diagnostic flags

<name>	Description
error=<flag>	Turn the warning <flag> into an error.
no-error=<flag>	Leave the warning <flag> as a warning even if `-Werror` is specified.
<flag>	Enable the warning <flag>.
no-<flag>	Suppress the warning <flag>. **Caution:** Reducing the severity of diagnostic messages might prevent the tool from reporting important faults. Arm recommends that you do not reduce the severity of diagnostics unless you understand the impact on your software.

For a list of the diagnostic flags, see Diagnostic flags in Clang. Arm® Compiler for Embedded supports only those diagnostic flags that are relevant to supported Arm Compiler for Embedded features.

See Controlling Errors and Warnings in the Clang Compiler User's Manual for full details about controlling diagnostics with `armclang`.


Note

The documentation at http://clang.llvm.org/docs is continually being updated, and might not be aligned with the Arm Compiler for Embedded version you are using. For older documents that might be a better match to your Arm Compiler for Embedded version, see https://releases.llvm.org.

Related information

Options for controlling diagnostics with armclang

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.1.119 -Wl

Specifies linker command-line options to pass to the linker when a link step is being performed after compilation.

See the armlink Command-line Options for information about available linker options.

Syntax

```
-Wl, <opt>,[<opt>[,...]]
```

Parameters

<opt>

A linker command-line option to pass to the linker.

You can specify a comma-separated list of options or `option=argument` pairs.

Restrictions

The linker generates an error if `-Wl` passes unsupported options.

Examples

The following examples show the different syntax usages. They are equivalent because `armlink` treats the single option `--list=diag.txt` and the two options `--list diag.txt` equivalently:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 hello.c -Wl,--split,--
list,diag.txt
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 hello.c -Wl,--split,--
list=diag.txt
```

Related information

-Xlinker on page 228

2.1.120 -Xlinker

Specifies linker command-line options to pass to the linker when a link step is being performed after compilation.

Syntax

```
-Xlinker <opt>
```

Parameters

<opt>

A linker command-line option to pass to the linker.

If you want to pass multiple options, use multiple `-Xlinker` options.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

See the armlink Command-line Options for information about available linker options.

Restrictions

The linker generates an error if `-Xlinker` passes unsupported options.

Examples

This example passes the option `--split` to the linker:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 hello.c -Xlinker --split
```

This example passes the options `--list diag.txt` to the linker:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 hello.c -Xlinker --list \
    -Xlinker diag.txt
```

Related information

-Wl on page 228

2.1.121 -x (armclang)

Specifies the language of source files.

Default

By default the compiler determines the source file language from the filename suffix, as follows:

- `.cpp`, `.cxx`, `.c++`, `.cc`, and `.CC` indicate C++, equivalent to `-x c++`.
- `.c` indicates C, equivalent to `-x c`.
- `.s` (lowercase) indicates assembly code that does not require preprocessing, equivalent to `-x assembler`.
- `.S` (uppercase) indicates assembly code that requires preprocessing, equivalent to `-x assembler-with-cpp`.



Note

Windows platforms do not detect `.s` files correctly because the file system does not distinguish case.

Syntax

```
-x <language>
```

Parameters

`<language>`

Specifies the language of subsequent source files, one of the following:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

c
C code.

c++
C++ code.

assembler-with-cpp
Assembly code containing C directives that require the C preprocessor.

assembler
Assembly code that does not require the C preprocessor.

Operation

`-x` overrides the default language standard for the subsequent input files that follow it on the command-line. For example:

```
armclang inputfile1.s -xc inputfile2.s inputfile3.s
```

In this example, `armclang` treats the input files as follows:

- `inputfile1.s` appears before the `-xc` option, so `armclang` treats it as assembly code because of the `.s` suffix.
- `inputfile2.s` and `inputfile3.s` appear after the `-xc` option, so `armclang` treats them as C code.


Note

Use `-std` to set the default language standard.

Related information

-D (armclang) on page 44
Preprocessing assembly code

2.1.122 -###

Displays the commands that invoke the compiler and sub-tools, such as `armlink`, without executing those commands.

Syntax

```
-###
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

The `-###` compiler option produces diagnostic output showing exactly how the compiler and linker are invoked, displaying the options for each tool. The `-###` compiler option also displays version information.

With the `-###` option, `armclang` only displays this diagnostic output. `armclang` does not compile source files or invoke `armlink`.


Note

To display the diagnostic output and execute the commands, use the `-v` option.

Related information

-v (armclang) on page 223

2.1.123 Examples for the armclang -ffp-contract command-line option

These examples show how you can use `-ffp-contract=<value>` and `#pragma STDC FP_CONTRACT <value>` together when compiling with `-ffp-model=std` to change the behavior of the compiler in different ways.

For more information about the `-ffp-contract` command-line option, see -ffp-contract.

About the -ffp-contract examples

The first three examples use the file `foo.c` that contains the following C code:

```
// A single C statement with both an FP multiply
// operation and an FP addition operation
float all_ops_in_same_line(float a, float b, float c)
{
    return a * b + c;
}

// Separate C statements for each FP operation
float ops_on_different_lines(float a, float b, float c)
{
    float temp = a * b;
    return temp + c;
}
```

Depending on how you compile this file, either one or both of the operations have FP contractions applied to them. The following A64 instructions are shown in the examples:

Instruction opcode	Operation description
`FADD`	Floating-point addition
`FMUL`	Floating-point multiplication
`FMADD`	Fused FP multiply-accumulate

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A fused operation is one where FP rounding is applied only once after all calculations are completed. A non-fused operation is where FP rounding is applied after each separate FP calculation performed by the instruction. Floating-point contraction uses fused operations when enabled. Floating-point operations that have had contraction applied to them are considered to be fused.

For more examples showing the interaction of the `FP_CONTRACT` pragma and the `-ffp-contract=<value>` option, see the examples for #pragma STDC FP_CONTRACT examples.

Example 1: Compiling with the default options

Compile `foo.c` with the following command:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -O1 -ffp-mode=std -c -S foo.c
 -o -
```

The output contains the following code:

```
...
all_ops_in_same_line:
        fmadd   s0, s0, s1, s2
        ret
...
ops_on_different_lines:
        fmul    s0, s0, s1
        fadd    s0, s0, s2
        ret
...
```

With `-ffp-mode=std`, `-ffp-contract=on` is the default. This means that only operations in the same line are fused. Therefore, the result of this example is as follows:

Function	Operation fused (Y/N)
`all_ops_in_same_line()`	Y
`ops_on_different_lines()`	N

Example 2: Compiling with -ffp-contract=off

Compile `foo.c` with the following command:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -O1 -ffp-mode=std -ffp-
contract=off -c -S foo.c -o -
```

The output contains the following code:

```
...
all_ops_in_same_line:
        fmul    s0, s0, s1
        fadd    s0, s0, s2
        ret
...
ops_on_different_lines:
        fmul    s0, s0, s1
        fadd    s0, s0, s2
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
        ret
...
```

Therefore, the outcome of this example is as follows:

Function	Operation fused (Y/N)
`all_ops_in_same_line()`	N
`ops_on_different_lines()`	N

Example 3: Compiling with -ffp-contract=fast

Compile `foo.c` with the following command:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -O1 -ffp-mode=std -ffp-
contract=fast -c -S foo.c -o -
```

The output is expected to contain the following code:

```
...
all_ops_in_same_line:
        fmadd   s0, s0, s1, s2
        ret
...
ops_on_different_lines:
        fmadd   s0, s0, s1, s2
        ret
...
```

Therefore, the result of this example is as follows:

Function	Operation fused (Y/N)
`all_ops_in_same_line()`	Y
`ops_on_different_lines()`	Y

Example 4: Keeping contraction enabled for some functions but not others

This example uses the file `bar.c` that contains the following C code:

```
float first_function(float a, float b, float c)
{
    return a * b + c;
}

float second_function(float a, float b, float c)
{
    return a * b + c;
}
```

This file contains two identical functions, both with a single C statement for all FP operations within the function. To configure FP contraction separately for these functions using only `-ffp-contract=<value>` is not sufficient. You must also use `#pragma STDC FP_CONTRACT <value>` in the source code.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

In this example, FP contraction is to be disabled for `first_function()`, but enabled for `second_function()`.

Compile `bar.c` with the following command:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -O1 -ffp-mode=std -ffp-
contract=off -c -S bar.c -o -
```

The output contains the following code:

```
...
first_function:
        fmul    s0, s0, s1
        fadd    s0, s0, s2
        ret
...
second_function:
        fmul    s0, s0, s1
        fadd    s0, s0, s2
        ret
...
```

Without any use of `#pragma STDC FP_CONTRACT <value>`, the result of this example is as follows:

Function	Operation fused (Y/N)
`first_function()`	N
`second_function()`	N

To enable FP contraction for `second_function()`, modify `bar.c` to the following:

```
float first_function(float a, float b, float c)
{
    return a * b + c;
}

#pragma STDC FP_CONTRACT ON

float second_function(float a, float b, float c)
{
    return a * b + c;
}
```

Re-compiling `bar.c` with the same options as before results in the following code:

```
first_function:
        fmul    s0, s0, s1
        fadd    s0, s0, s2
        ret
...
second_function:
        fmadd   s0, s0, s1, s2
        ret
```

Therefore, observe that the desired result is as follows:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Function	Operation fused (Y/N)
`first_function()`	N
`second_function()`	Y

Related information

-ffp-mode on page 62
#pragma STDC on page 329
A64 -- SVE Instructions (alphabetic order)

2.1.124 Examples for the armclang -fsanitize command-line option

These examples show how to use the `memtag-stack` and `function` sanitizers.

For more information about the `-fsanitize` command-line option, see -fsanitize, -fno-sanitize.

Example: Effect of the -fsanitize=memtag-stack option

Create the source file `func.c` containing the following code:

```
extern void func2 (int* a);

void func1(void)
{
    int x=10;
    int y=20;

    func2(&x);
    func2(&y);
}
```

Compile `func.c`, without memory tagging stack protection, using the following command line:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a+memtag -S -O1 func.c -o
 mem_no_protect.s
```

The generated assembly file `mem_no_protect.s` contains the following code:

```
func1:
        str     x30, [sp, #-16]!        // 8-byte Folded Spill
        mov     w8, #10
        mov     w9, #20
        add     x0, sp, #12             // =12
        stp     w9, w8, [sp, #8]
        bl      func2
        add     x0, sp, #8              // =8
        bl      func2
        ldr     x30, [sp], #16          // 8-byte Folded Reload
        ret
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Compile `func.c`, with memory tagging stack protection, using the following command line:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a+memtag -S -O1 func.c -
fsanitize=memtag-stack -o mem_with_protect.s
```

The generated assembly file `mem_with_protect.s` contains the following code:

```
func1:
        sub     sp, sp, #48
        stp     x30, x19, [sp, #32]     // 16-byte Folded Spill
        irg     x19, sp
        mov     w8, #10
        mov     w9, #20
        addg    x0, x19, #16, #1
        stgp    x8, xzr, [x0]
        stgp    x9, xzr, [x19]
        bl      func2
        mov     x0, x19
        bl      func2
        st2g    sp, [sp], #32
        ldp     x30, x19, [sp], #16     // 16-byte Folded Reload
        ret
```

When using the `-fsanitize=memtag-stack` option:

- The compiler generates memory tagging instructions, for example `ADDG`, `IRG`, `STGP`, and `ST2G`, to ensure that the pointers and the variables on the stack are tagged. For information on these instructions, see the base instructions in Arm A64 Instruction Set Architecture.
- The compiler uses an extra 32 bytes of memory on the stack for the variables in `func.c`, whose addresses are taken.

Example: Effect of the -fsanitize=function option

Create the source file `foo.c` containing the following code:

```
#include <stdio.h>

extern int bar(int a, int b);

int foo(int a)
{
    return a + 0x11223344;
}

int main(void)
{
    puts("Hello, world!");

    bar(1, 2);

    puts("Goodbye, world!");

    return 0;
}

void __ubsan_handle_add_overflow(void)
{
    puts("Add overflow occurred :(");
}

void __ubsan_handle_function_type_mismatch(void)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
{
    puts("Function pointer type mismatch :(");
}
```

Create the source file `bar.c` containing the following code:

```
extern int foo(int a, int b);

int (*foo_ptr)(int a, int b);

int bar(int a, int b)
{
    foo_ptr = &foo; /* The compiler cannot see the implementation of
                       foo() when compiling bar.c, so it cannot verify
                       if the function pointer and the function
                       prototype match */
    return foo_ptr(a, b);
}
```

Compile and link with:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O1 -mthumb -fsanitize=function -
c foo.c bar.c
armlink foo.o bar.o -o image.axf
```

Run the image on a suitable target or *Fixed Virtual Platform* (FVP) model. The following messages are output:

```
Hello, world!
Function pointer type mismatch :(
Goodbye, world!
```

Compile and link with `-S` to generate the disassembly for each C file, `foo.s` and `bar.s`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O1 -mthumb -fsanitize=function -
S foo.c bar.c
```

`foo.s` contains:

```
...
    .type   foo,%function
    .long   3238382334                      @ 0xc105cafe   [1]
    .long   329620                          @ 0x50794      [2]
    .code   16
    .thumb_func
foo:
...
    .section     .text.main,"ax",%progbits
    .hidden  main
    .globl   main
    .p2align     2
    .type    main,%function
    .long    3238382334                       @ 0xc105cafe [3]
    .long    917620134                        @ 0x36b1c5a6 [4]
    .code    16
    .thumb_func
main:
...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    .section    .text.__ubsan_handle_add_overflow,"ax",%progbits
    .hidden __ubsan_handle_add_overflow     @ -- Begin function
 __ubsan_handle_add_overflow
    .globl  __ubsan_handle_add_overflow
    .p2align    1
    .type    __ubsan_handle_add_overflow,%function
    .long    3238382334                     @ 0xc105cafe [5]
    .long    2772461324                     @ 0xa540670c [6]
    .code    16
    .thumb_func
__ubsan_handle_add_overflow:
    .fnstart
    .cfi_startproc
...
    .section    .text.__ubsan_handle_function_type_mismatch,"ax",%progbits
    .hidden    __ubsan_handle_function_type_mismatch @ -- Begin function
 __ubsan_handle_function_type_mismatch
    .globl    __ubsan_handle_function_type_mismatch
    .p2align    2
    .type    __ubsan_handle_function_type_mismatch,%function
    .long     3238382334                     @ 0xc105cafe [7]
    .long     2772461324                     @ 0xa540670c [8]
    .code     16
    .thumb_func
__ubsan_handle_function_type_mismatch:
    .fnstart
    .cfi_startproc
@ %bb.0:
    adr r0, .LCPI3_0
    b   puts
    .p2align    2
@ %bb.1:
.LCPI3_0:
.L.str.4:
    .asciz  "Function pointer type mismatch :(\000\000"
.Lfunc_end3:
    .size    __ubsan_handle_function_type_mismatch, .Lfunc_end3-
__ubsan_handle_function_type_mismatch
    .cfi_endproc
    .cantunwind
    .fnend
```

`bar.s` contains:

```
...
    .file   "bar.c"
    .section    .text.bar,"ax",%progbits
    .hidden bar
    .globl  bar
    .p2align    1
    .type   bar,%function
    .long   3238382334                      @ 0xc105cafe   [9]
    .long   1457894821                      @ 0x56e5b5a5   [10]
    .code   16
    .thumb_func
bar:
    .fnstart
    .cfi_sections .debug_frame
    .cfi_startproc
...
    movw    r0, :lower16:foo_ptr
    movt    r0, :upper16:foo_ptr            @ r0 = address of foo_ptr
    movw    r1, :lower16:foo
    movt    r1, :upper16:foo                @ r1 = address of foo
    str     r1, [r0]
    bic     r0, r1, #1                      @ clear bit 0 of r1 and store in r0
    ldr     r1, [r0, #-8]                   @ load r1 from the word 8 bytes below
 the address in r0 (0xc105cafe)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    movw    r2, #51966
    movt    r2, #49413                      @ [11] r2 = 0xc105cafe
    cmp     r1, r2
    bne     .LBB0_2                         @ r1 = r2, so continue
@ %bb.1:
    ldr     r0, [r0, #-4]                   @ load r0 from the word 4 bytes below
 the address in r0 (0x50794)
    movw    r1, #46501
    movt    r1, #22245                      @ [12] r1 = 0x56e5b5a5
    cmp     r0, r1
    bne     .LBB0_3                         @ r0 NE r1, so branch to .LBB0_3 and
 then to
                                            @ the function_type_mismatch handler
.LBB0_2:
    mov     r0, r5
    mov     r1, r4
    pop.w   {r4, r5, r7, lr}
    b    foo
.LBB0_3:
    movw    r0, :lower16:.L__unnamed_1
    movt    r0, :upper16:.L__unnamed_1
    movw    r1, :lower16:foo
    movt    r1, :upper16:foo
    bl   __ubsan_handle_function_type_mismatch
    b   .LBB0_2
...
foo_ptr:
    .long    0
    .size    foo_ptr, 4
...
    .type    .L__unnamed_2,%object           @ @0
    .section     .rodata..L__unnamed_2,"a",%progbits
    .p2align     4, 0x0
.L__unnamed_2:
    .short  65535                           @ 0xffff
    .short  0                               @ 0x0
    .asciz  "'int (*)(int, int)'"
    .size   .L__unnamed_2, 24

    .type   .L__unnamed_1,%object           @ @1
    .section    .data..L__unnamed_1,"aw",%progbits
    .p2align    2, 0x0
.L__unnamed_1:
    .long   .L.src
    .long   11                              @ 0xb
    .long   12                              @ 0xc
    .long   .L__unnamed_2
    .size   .L__unnamed_1, 16
...
```

Notes on `foo.s` and `bar.s`

The data that the function sanitizer stores immediately before the start of each function consists of two 32-bit words.

The value of the first 32-bit word is the same for all functions `0xc105cafe`. See the values at [1], [3], [5], [7], and [9]. This value is checked at [11] to see if the file containing the target function is compiled with `-fsanitize=function`.

If the first value is present, then the value of the second 32-bit word is checked at [12]. This value depends on the following:

- The function return type.
- The number of function arguments.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The type of the function arguments.

Compare the values at [2], [4], [6], [8], and [10].

When you compile `bar.c`, the compiler generates a value for the second 32-bit word based on the definition of `foo()` in `bar.c`. Because this definition is different from that in `foo.c`, the value at [12] does not match the value from `foo.c`. Therefore, `bar()` calls the handler function `ubsan_handle_function_type_mismatch()`.

2.1.125 Examples for the armclang -mbranch-protection command-line option

These examples show how to use the `-mbranch-protection` command-line option.

For more information about this option, see -mbranch-protection.

Example: -mbranch-protection=bti

Branch Target Identification (BTI) is an architectural hardware security feature that limits the destination addresses of indirect branch instructions. This feature can help to prevent *Jump-Oriented Programming* (JOP) style security attacks.

If you compile with BTI enabled, the compiler adds a `BTI` instruction at the start of each function. The specific instruction encoding used depends on several factors. For more information, see *Hint space instructions versus non-hint space instructions for branch protection*.

For Arm®v8-M targets with the Main Extension, BTI is implemented using the *Pointer Authentication Code and Branch Target Identification* (PACBTI) extension.

For AArch64 state, BTI is implemented using FEAT_BTI. This implementation means that indirect branch instructions, for example `BR`, can be taken only to an address that contains a `BTI` instruction.

Create the file `bti.c` containing the following code:

```
extern int secure_function(void);

int func(int a, int b)
{
    return a + b + secure_function();
}
```

Compile the `bti.c` file for AArch64 state using the following command:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -S -O1 -mbranch-
protection=bti bti.c
```

The generated assembly file, `bti.s`, contains the `BTI` instruction at the start of `func()`:

```
func:
        bti     c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
        stp     x30, x19, [sp, #-16]!
        add     w19, w1, w0
        bl      secure_function
        add     w0, w19, w0
        ldp     x30, x19, [sp], #16
        ret
```

The `BTI` instruction at the start of `func()` is the only valid destination of an indirect branch instruction, such as `BR`. It is not possible for an indirect branch instruction to jump to any other instruction within `func()`. A target that implements FEAT_BTI triggers an architectural exception if an attempt is made to do so.

For more information, see the following documents:

- Jump-oriented programming in *Learn the architecture - Providing protection for complex software*.
- Branch Target Identification in *Armv8.1-M PACBTI Extensions*.

Example: -mbranch-protection=pac-ret

Pointer authentication codes are part of an architectural hardware security feature that can determine if the return address of a function has been modified. Use of *Pointer Authentication Code* (PAC) branch protection can help to prevent *Return-Oriented Programming* (ROP) style security attacks.

If you compile with PAC branch protection enabled, the compiler generates instructions that perform the following actions:

- On function entry, sign the return address in LR with a PAC.
- Before function return, check whether LR has been modified.

The specific instructions used depend on several factors. For more information, see *Hint space instructions versus non-hint space instructions for branch protection*.

For AArch64 state, PAC branch protection is implemented using the following features:

- The pointer authentication (FEAT_PAuth) feature.
- The pointer authentication instructions that allow signing of LR using SP and PC as diversifiers (FEAT_PAuth_LR).

For Armv8-M targets with the Main Extension, PAC branch protection is implemented using the PACBTI extension.

Create the `pac.c` file containing the following code:

```
extern int secure_function(void);

int func(int a, int b)
{
    return a + b + secure_function();
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Compile the `pac.c` file for AArch64 state using the following command line:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -S -O1 -mbranch-
protection=pac-ret pac.c
```

The generated assembly file, `pac.s`, contains:

- The `PACIASP` instruction at the start of `func()` to sign LR using SP as a diversifier.
- The `RETAA` instruction instead of `RET` to perform a PAC check before returning from `func()`.

```
func:
        paciasp
        stp     x30, x19, [sp, #-16]!
        add     w19, w1, w0
        bl      secure_function
        add     w0, w19, w0
        ldp     x30, x19, [sp], #16
        retaa
```

These instructions mean that a target that implements FEAT_PAuth triggers an architectural exception if any of the following conditions are true:

- SP has not been restored to its original value at the start of `func()`.
- LR does not contain the valid PAC generated by the `PACIASP` instruction at the start of `func()`.

For more information, see the following documents:

- Return-oriented programming in *Learn the architecture - Providing protection for complex software*.
- Pointer Authentication Code in *Armv8.1-M PACBTI Extensions*.

Example: -mbranch-protection=standard

This table shows when the following types of branch protection are enabled by default when you compile with `-mbranch-protection=standard`:

- BTI
- PAC using only the *Stack Pointer* (SP) as a diversifier
- PAC with *Program Counter* (PC) as the second diversifier in addition to SP as the first diversifier

The table is based on the target architecture, and whether the command-line options specify if the target includes pointer authentication instructions that allow signing of the *Link Register* (LR) using SP and PC as diversifiers (FEAT_PAuth_LR).

Table 2-37: -mbranch-protection=standard behavior

Target	BTI enabled	PAC enabled	PC as the second diversifier enabled	Equivalent `-mbranch-protection=<protection>` option
AArch64 with FEAT_PAuth_LR	Yes	Yes	Yes	`bti+pac-ret+pc`
AArch64 without FEAT_PAuth_LR	Yes	Yes	No	`bti+pac-ret`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Target	BTI enabled	PAC enabled	PC as the second diversifier enabled	Equivalent `-mbranch-protection=<protection>` option
Armv8-M	Yes	Yes	-	`bti+pac-ret`

For more information about the type of instructions generated, see *Hint space instructions versus non-hint space instructions for branch protection*.

Hint space instructions versus non-hint space instructions for branch protection

The Arm architecture implementation of BTI and PAC provides hint space instructions and non-hint space instructions that are used to perform branch protection.

The term "hint space" refers to an instruction encoding space in the Arm architecture specification that is reserved for instructions that the processor can safely ignore. For example, if an instruction `I` is implemented as a hint space instruction as part of an architectural feature `F` in version `V` of the architecture, all the following statements are true:

- A processor based on a version equal to or later than `V` that implements `F` executes `I`.
- A processor based on a version equal to or later than `V` that does not implement `F` ignores `I` and treats it as a `NOP` instruction.
- A processor based on a version earlier than `V` ignores `I` and treats it as a `NOP` instruction.

Hint space instructions enable you to write software that is backwards compatible with earlier architecture versions and processors that do not have hardware support for the instruction.

For BTI and PAC, this means that you can use the compiler to generate code that contains hint space instructions that run on any target regardless of whether it has hardware support for branch protection features.

However, hint space instructions are sometimes less efficient than non-hint space instructions. If you compile code that does not need to be backwards compatible, you can generate code containing non-hint space instructions that can only run on a target with hardware support for branch protection features.

The choice of hint space or non-hint space instructions depends on whether the target options specified to the compiler enable the following features:

For AArch64 state:

- The Pointer authentication (FEAT_PAuth) feature.
- The Pointer authentication instructions that allow signing of LR using SP and PC as diversifiers (FEAT_PAuth_LR).
- The BTI (FEAT_BTI) feature.

For an Armv8-M target with the Main Extension:

- The PACBTI extension.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Tip

When compiling for AArch64 state, you can compile with `--print-enabled-features` to check if the target options enable a particular PAC or BTI feature. If the output from this option lists the feature identifier (`FEAT_<name>`) for a particular feature, then the target options enable the feature.

The following tables list when different types of instructions are generated for adding branch protection to compiled code:

Table 2-38: Hint space instructions versus non-hint space instructions for branch protection in AArch64 state

FEAT_PAuth	FEAT_PAuth_LR	FEAT_BTI	PC as the second diversifier enabled	PAC instruction type	BTI instruction type
Included	Included	Included	Yes	Non-hint space	Non-hint space
Included	Not included	Included	Yes	Hint space (FEAT_PAuth_LR), and non-hint space (FEAT_PAuth)	Non-hint space
Included	Not included	Not included	Yes	Hint space (FEAT_PAuth_LR), and non-hint space (FEAT_PAuth)	Hint space
Included	Not included	Included	No	Non-hint space	Non-hint space
Included	Not included	Not included	No	Non-hint space	Hint space
Not included	Not included	Included	-	Hint space	Non-hint space
Not included	Not included	Not included	-	Hint space	Hint space

Table 2-39: Hint space instructions versus non-hint space instructions for branch protection in Armv8-M targets with the Main Extension

PACBTI extension	PAC instruction type	BTI instruction type
Included	Non-hint space	Non-hint space
Not included	Hint space	Hint space

2.2 Compiler-specific Keywords and Operators

Keywords and operators that are specific to Arm® Compiler for Embedded, and that are extensions to the C and C++ Standards.

Keyword extensions

The Arm Compiler for Embedded compiler, `armclang`, provides keywords that are extensions to the C and C++ Standards.

Standard C and Standard C++ keywords that do not have behavior or restrictions specific to `armclang` are not documented.

Keyword extensions that `armclang` supports:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- __alignof__
- __asm
- __declspec (deprecated)
- __inline

Related information

Global named register variables on page 252

2.2.1 __alignof__

The `__alignof__` keyword enables you to inquire about the alignment of a type or variable.


Note

This keyword is a GNU compiler extension that `armclang` supports.

Syntax

```
__alignof__(<type>)
__alignof__(<expr>)
```

Parameters

`<type>`

A data type.

`<expr>`

An lvalue, such as a variable name.

An rvalue, such as __alignof__(42).

Returns

`__alignof__(<type>)` returns the alignment requirement for the 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: Alignment requirements for a variety of data types

Create the file `alignof.c` containing:

```
#include <stdio.h>

int main(void)
{
  int       var_i;
  char      var_c;
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
  double    var_d;
  float     var_f;
  long      var_l;
  long long var_ll;

  printf("Alignment requirement from data type:\n");
  printf("  int       : %d\n", __alignof__(int));
  printf("  char      : %d\n", __alignof__(char));
  printf("  double    : %d\n", __alignof__(double));
  printf("  float     : %d\n", __alignof__(float));
  printf("  long      : %d\n", __alignof__(long));
  printf("  long long : %d\n", __alignof__(long long));
  printf("\n");
  printf("Alignment requirement from data type of lvalue:\n");
  printf("  int       : %d\n", __alignof__(var_i));
  printf("  char      : %d\n", __alignof__(var_c));
  printf("  double    : %d\n", __alignof__(var_d));
  printf("  float     : %d\n", __alignof__(var_f));
  printf("  long      : %d\n", __alignof__(var_l));
  printf("  long long : %d\n", __alignof__(var_ll));
}
```

To compile, enter:

```
armclang --target=arm-arm-none-eabi -march=armv8-a alignof.c -o alignof.axf
```

Run the `alignof.axf` image on a suitable target. The output is:

```
Alignment requirement from data type:

  int       : 4
  char      : 1
  double    : 8
  float     : 4
  long      : 4
  long long : 8

Alignment requirement from data type of lvalue:
  int       : 4
  char      : 1
  double    : 8
  float     : 4
  long      : 4
  long long : 8
```

Related information

Alignment support in Arm Compiler for Embedded 6

2.2.2 __asm

This keyword passes information to the `armclang` assembler.

Syntax

```
__asm(<code>)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<code>

The inline assembly code.

Operation

The precise action of this keyword depends on its usage. To prevent the compiler from optimizing the code out, use `__asm volatile (...)`.

The general form of an `__asm` inline assembly statement is described in Writing inline assembly code.

Example: Using __asm for inline assembly code

The `__asm` keyword can incorporate inline GCC syntax assembly code into a function. For example:

```
#include <stdio.h>

int add(int i, int j)
{
  int res = 0;
  __asm(
    "ADD %[result], %[input_i], %[input_j]"
    : [result] "=r" (res)
    : [input_i] "r" (i), [input_j] "r" (j)
  );
  return res;
}

int main(void)
{
  int a = 1;
  int b = 2;
  int c = 0;

  c = add(a,b);

  printf("Result of %d + %d = %d\n", a, b, c);
}
```

Example: Using __asm for embedded assembly code

For embedded assembly, you cannot use the `__asm` keyword on the function declaration. Use the `__attribute__((naked))` function attribute on the function declaration. For more information, see __attribute__((naked)) function attribute. For example:

```
__attribute__((naked)) void foo (int i);
```

Naked functions with the `__attribute__((naked))` function attribute only support assembler instructions in the basic format:

```
__asm(<code>);
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Using __asm to specify an assembly label for a C symbol

The `__asm` keyword can specify an assembly label for a C symbol. For example:

```
int count __asm("count_v1"); // export count_v1, not count
```

Related information

__attribute__((naked)) function attribute on page 267

2.2.3 __declspec attributes

The `__declspec` keyword enables you to specify special attributes of objects and functions.



Note

The `__declspec` keyword is deprecated. Use the `__attribute__` function attribute.

The `__declspec` keyword must prefix the declaration specification. For example:

```
__declspec(noreturn) void overflow(void);
```

The following `__declspec` attributes are available:

- __declspec(noinline)
- __declspec(noreturn)
- __declspec(nothrow)

`__declspec` attributes are storage class modifiers. They do not affect the type of a function or variable.

2.2.4 __declspec(noinline)

The `__declspec(noinline)` attribute suppresses the inlining of a function at the call points of the function.



Note

- The `__declspec` keyword is deprecated.
- This `__declspec` attribute has the function attribute equivalent `__attribute__((noinline))`.

Syntax

```
__declspec(noinline)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Example

```
/* Suppress inlining of foo() wherever foo() is called */
__declspec(noinline) int foo(void);
```

Related information

__attribute__((noinline)) function attribute on page 268

2.2.5 __declspec(noreturn)

The `__declspec(noreturn)` attribute asserts that a function never returns.


Note

- The `__declspec` keyword is deprecated.
- This `__declspec` attribute has the function attribute equivalent `__attribute__((noreturn))`.

Syntax

```
__declspec(noreturn)
```

Parameters

None.

Restrictions

The return address is not preserved when calling the `noreturn` function. This limits the ability of a debugger to display the call stack.

Operation

Use this attribute to reduce the cost of calling a function that never returns, such as `exit()`. If a `noreturn` function returns to its caller, the behavior is undefined.

Example

```
__declspec(noreturn) void overflow(void); // never return on overflow
int negate(int x)
{
    if (x == 0x80000000) overflow();
    return -x;
}
```

Related information

__attribute__((noreturn)) function attribute on page 271

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.2.6 __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.


Note

- The `__declspec` keyword is deprecated.
- This `__declspec` attribute has the function attribute equivalent `__attribute__((nothrow))`.

Syntax

```
__declspec(nothrow)
```

Parameters

None.

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.

Operation

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.

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`.

Example

```
struct S
{
    ~S();
};
__declspec(nothrow) extern void f(void);
void g(void)
{
    S s;
    f();
}
```

Related information

__attribute__((nothrow)) function attribute on page 272
Standard C++ library implementation definition

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.2.7 __inline

The `__inline` keyword suggests to the compiler that it compiles a C or C++ function inline, if it is sensible to do so.

Syntax

```
__inline
```

Parameters

None.

Operation

`__inline` can be used in C90 code, and serves as an alternative to the C99 `inline` keyword.

Both `__inline` and `__inline__` are supported in `armclang`.

Example

```
static __inline int f(int x){
    return x*5+1;
}

int g(int x, int y){
    return f(x) + f(y);
}
```

Related information

Inline functions on page 341

2.2.8 __promise

`__promise` 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

```
__promise(<expr>)
```

Parameters

`<expr>`

An expression that evaluates to nonzero.

Operation

You must `#include <assert.h>` to use `__promise(<expr>)`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If assertions are enabled (by not defining `NDEBUG`) and the macro `__DO_NOT_LINK_PROMISE_WITH_ASSERT` is not defined, then the promise is checked at runtime by evaluating `<expr>` as part of `assert(<expr>)`.

2.2.9 __unaligned

The `__unaligned` keyword is a type qualifier that tells the compiler to treat the pointer or variable as an unaligned pointer or variable.

Syntax

```
__unaligned
```

Parameters

None.

Operation

Members of packed structures might be unaligned. Use the `__unaligned` keyword on pointers that you use for accessing packed structures or members of packed structures.

Example: __unaligned assignment without type casting

```
typedef struct __attribute__((packed)) S {
    char c;
    int x;
};

int f1_load(__unaligned struct S *s)
{
    return s->x;
}
```

The compiler generates an error if you assign an unaligned pointer to a regular pointer without type casting.

Example: __unaligned with and without type casting

```
struct __attribute__((packed)) S { char c; int x; };

void foo(__unaligned struct S *s2)
{
    int *p = &s2->x;             // compiler error because &s2->x is an unaligned
 pointer but p is a regular pointer.
    __unaligned int *q = &s2->x; // No error because q and &s2->x are both unaligned
 pointers.
}
```

Related information

Alignment support in Arm Compiler for Embedded 6

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.2.10 Global named register variables

The compiler enables you to use the `register` storage class specifier to store global variables in general-purpose registers. These variables are called global named register variables.

Syntax

```
register <Type> <VariableName> __asm("<Reg>")
```

Parameters

`<Type>`

The data type of variable. The data type can be `char` or any 8-bit, 16-bit, or 32-bit integer type, or their respective pointer types.

`<VariableName>`

The name of the variable.

`<Reg>`

The general-purpose register to use to store the variable. The general purpose register can be `R6` to `R11`.

Restrictions

This feature is only available for AArch32 state.

If you use `-mpixolib`, then you must not use the following registers as global named register variables:

- R8
- R9

If you use `-frwpi` or `-frwpi-lowering`, then you must not use register R9 as a global named register variable. If you do, then `armclang` throws an error.

If needed, the Arm® ABI reserves the following registers for use as a frame pointer:

- R7 in T32 state.
- R11 in A32 state.

`armclang` reports an error if you use global named register variables under these conditions.

Code size

Declaring a general-purpose register as a global named register variable means that the register is not available to the compiler for other operations. If you declare too many global named register variables, code size increases significantly. Sometimes, your program might not compile, for example if there are insufficient registers available to compute a particular expression.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

Using global named register variables enables faster access to these variables than if they are stored in memory.


Note

For correct runtime behavior:

- You must use the relevant `-ffixed-r<N>` option for all the registers that you use as a global named register variable.
- You must use the relevant `-ffixed-r<N>` option to compile any source file that contains calls to external functions that use global named register variables.

For example, to use register R8 as a global named register for an integer `foo`, you must use:

```
register int  foo  __asm("R8")
```

For this example, you must compile with the command-line option `-ffixed-r8`. For more information, see -ffixed-r<N>.

The Arm standard library has not been built with any `-ffixed-r<N>` option. If you want to link application code containing global named register variables with the Arm standard library, then:

- To ensure correct runtime behavior, ensure that the library code does not call code that uses the global named register variables in your application code.
- The library code might push and pop the register to stack, even if your application code uses this register as a global named register variable.


Note

- If you use the register storage class, then you cannot use any additional storage class such as `extern` or `static` for the same variable.
- In C and C++, global named register variables cannot be initialized at declaration.

Examples

The following example demonstrates the use of `register` variables and the relevant `-ffixed-r<N>` option.

Source file `main.c` contains the following code:

```
#include <stdio.h>

/* Function defined in another file that will be compiled with
   -ffixed-r8 -ffixed-r9. */
extern int add_ratio(int a, int b, int c, int d, int e, int f);

/* Helper variable */
int other_location = 0;

/* Named register variables */
register int foo __asm("r8");
register int *bar __asm("r9");
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
__attribute__((noinline))  int initialise_named_registers(void)
{
  /* Initialise pointer-based named register variable */
  bar = &other_location;

  /* Test using named register variables */
  foo = 1000;
  *bar = *bar + 1;
  return 0;
}

int main(void)
{
  initialise_named_registers();
  add_ratio(10, 2, 30, 4, 50, 6);
  printf("foo: %d\n", foo);  // expects to print 1000
  printf("bar: %d\n", *bar); // expects to print 1
}
```

Source file `sum.c` contains the following code:

```
/* Arbitrary function that could normally result in the compiler using R8 and R9.

When compiling with -ffixed-r8 -ffixed-r9, the compiler must not use registers
R8 or R9 for any function in this file.
*/
__attribute__((noinline)) int add_ratio(int a, int b, int c, int d, int e, int f)
{
  int sum;
  sum = a/b + c/d + e/f;
  if (a > b && c > d)
    return sum*e*f;
  else
    return (sum/e)/f;
}
```

Compile `main.c` and `sum.c` separately before linking them. This application uses global named register variables using R8 and R9, and therefore both source files must be compiled with the relevant `-ffixed-r<N>` option:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O2 -ffixed-r8 -ffixed-r9 -c
 main.c -o main.o --save-temps
armclang --target=arm-arm-none-eabi -march=armv8-a -O2 -ffixed-r8 -ffixed-r9 -c
 sum.c -o sum.o --save-temps
```

Link the two object files using `armlink`:

```
armlink --cpu=8-a.32 main.o sum.o -o image.axf
```

The use of the `armclang` option `--save-temps` enables you to look at the generated assembly code. The file `sum.s` is generated from `sum.c`, and does not use registers R8 and R9 in the `add_ratio()` function:

```
add_ratio:
...
    push    {r4, r5, r11, lr}
    .cfi_def_cfa_offset 16
    .cfi_offset lr, -4
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    .cfi_offset r11, -8
    .cfi_offset r5, -12
    .cfi_offset r4, -16
    ldr    r12, [sp, #20]
    sdiv    r4, r2, r3
    ldr    lr, [sp, #16]
    sdiv    r5, r0, r1
    add    r4, r4, r5
    cmp    r0, r1
    sdiv    r5, lr, r12
    cmpgt    r2, r3
    add    r4, r4, r5
    bgt    .LBB0_2
    sdiv    r0, r4, lr
    sdiv    r0, r0, r12
    pop    {r4, r5, r11, pc}
.LBB0_2:
    mul    r0, r12, lr
    mul    r0, r0, r4
    pop    {r4, r5, r11, pc}
```

The file `main.s` has been generated from `main.c`, and uses registers R8 and R9 only for the code that directly uses these global named register variables:

```
initialise_named_registers:
...
    movw    r9, :lower16:other_location
    mov    r8, #1000
    movt    r9, :upper16:other_location
    ldr    r0, [r9]
    add    r0, r0, #1
    str    r0, [r9]
    mov    r0, #0
    bx    lr
```

```
main:
...
    push    {r11, lr}
    .cfi_def_cfa_offset 8
    .cfi_offset lr, -4
    .cfi_offset r11, -8
    .pad    #8
    sub    sp, sp, #8
    .cfi_def_cfa_offset 16
    bl    initialise_named_registers
    mov    r0, #6
    mov    r1, #50
    str    r1, [sp]
    mov    r1, #2
    str    r0, [sp, #4]
    mov    r0, #10
    mov    r2, #30
    mov    r3, #4
    bl    add_ratio
    adr    r0, .LCPI1_0
    mov    r1, r8
    bl    __2printf
    ldr    r1, [r9]
    adr    r0, .LCPI1_1
    bl    __2printf
    mov    r0, #0
    add    sp, sp, #8
    pop    {r11, pc}
    .p2align    2
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

The Arm standard library code, such as the library implementations for the `printf()` function, might still use R8 and R9 because the standard library has not been built with any `-ffixed-r<N>` option.

Related information

-ffixed-r<N> on page 57

2.3 Compiler-specific Function, Variable, and Type Attributes

Function, variable, and type attributes that are specific to Arm® Compiler for Embedded, and that are extensions to the C and C++ Standards.

The `__attribute__` keyword enables you to specify special attributes of variables, structure fields, functions, and types.

The keyword format is either of the following:

```
__attribute__((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

Function attributes

The available function attributes are:

Table 2-40: Function attributes that the compiler supports, and their equivalents

Function attribute	Non-attribute equivalent
__attribute__((always_inline))	-
__attribute__((cmse_nonsecure_call))	-
__attribute__((cmse_nonsecure_entry))	-
__attribute__((const))	-
__attribute__((constructor(priority)))	-
__attribute__((format_arg(string-index)))	-
__attribute__((interrupt("type")))	-
__attribute__((malloc))	-
__attribute__((naked))	-
__attribute__((noinline))	`__declspec(noinline)`
__attribute__((nomerge)) [COMMUNITY]	-
__attribute__((nonnull))	-
__attribute__((no_instrument_function)) [COMMUNITY]	-
__attribute__((no_sanitize("option")))	-
__attribute__((noreturn))	`__declspec(noreturn)`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Function attribute	Non-attribute equivalent
__attribute__((nothrow))	`__declspec(nothrow)`
__attribute__((not_tail_called)) [COMMUNITY]	-
__attribute__((optnone))	-
__attribute__((pcs("calling_convention")))	-
__attribute__((pure))	-
__attribute__((section("name")))	-
__attribute__((target("options")))	-
__attribute__((unused))	-
__attribute__((used))	-
__attribute__((value_in_regs))	-
__attribute__((visibility("visibility_type")))	-
__attribute__((weak))	-
__attribute__((weakref("target")))	-

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))`.

Type attributes

The available type attributes are:

- __attribute__((aligned))
- __attribute__((packed))
- __attribute__((transparent_union))

Variable attributes

The available variable attributes are:

- __attribute__((alias))
- __attribute__((aligned("x")))
- __attribute__((common)), __attribute__((nocommon))
- __attribute__((deprecated))
- __attribute__((packed))
- __attribute__((section("name")))
- __attribute__((tls_model("model")))
- __attribute__((uninitialized))

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- __attribute__((unused))
- __attribute__((used))
- __attribute__((visibility("visibility_type")))
- __attribute__((weak))
- __attribute__((weakref("target")))

Example: Function attribute

```
int my_function(int b) __attribute__((const));
static int my_variable __attribute__((__unused__));
```

Example: Type attribute

```
typedef union {
    int i;
    float f;
} U __attribute__((transparent_union));
```

Example: Variable attribute

```
static int b __attribute__((__unused__));
```

2.3.1 __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.

Syntax

```
__attribute__((always_inline)) <type> <function>;
```

Parameters

None.

Restrictions

In some circumstances, the compiler might choose to ignore `__attribute__((always_inline))`, and not inline the function. For example:

- A recursive function is never inlined into itself.
- Functions that use `alloca()` or variable-length arrays might not be inlined.
- If the address of the function is assigned in the source code.
- If there is a call to the function from a different compilation unit without *Link-Time Optimization* (LTO).

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

Including `__attribute__((optnone))` with `__attribute__((always_inline))` prevents inlining.

The `-Rpass-missed=inline` option prints informative messages when the compiler ignores `__attribute__((always_inline))`.

Example: Using the `inline` and `__inline__` keywords with __attribute__((always_inline))

```
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
}
```

`__attribute__((always_inline))` does not affect the linkage characteristics of the function in the same way that the `inline` function-specifier does. When using `__attribute__((always_inline))`, if you want the declaration and linkage of the function to follow the rules of the `inline` function-specifier of the source language, then you must also use the keyword `inline` or `__inline__` (for C90). For example:

```
inline int max(int x, int y) __attribute__((always_inline));

int max(int x, int y)
{
    return x > y ? x : y; // always inline if possible
}
```

Related information

-Rpass on page 213
__attribute__((noinline)) function attribute on page 268
Inline functions on page 341
Selecting optimization options
Inlining functions

2.3.2 __attribute__((cmse_nonsecure_call)) function attribute

Declares a non-secure function type.

Syntax

C code:

```
void __attribute__((cmse_nonsecure_entry)) entry_func(int val)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

C++ code:

```
extern "C" void __attribute__((cmse_nonsecure_entry)) entry_func(int val)
```

Parameters

None.

Operation

A call to a function that switches state from Secure to Non-secure is called a non-secure function call. A non-secure function call can only happen through function pointers. This is a consequence of separating secure and non-secure code into separate executable files.

A non-secure function type must only be used as a base type of a pointer.

Example: Using the cmse_nonsecure_call attribute

```
#include <arm_cmse.h>

typedef void __attribute__((cmse_nonsecure_call)) nsfunc(void);

void default_callback(void) { ... }

// fp can point to a secure function or a non-secure function
nsfunc *fp = (nsfunc *) default_callback;   // secure function pointer

void __attribute__((cmse_nonsecure_entry)) entry(nsfunc *callback) {
    fp = cmse_nsfptr_create(callback);      // non-secure function pointer
}

void call_callback(void) {
    if (cmse_is_nsfptr(fp)){
        fp();           // non-secure function call
    }
    else {
        ((void (*)(void)) fp)();           // normal function call
    }
}
```

Related information

__attribute__((cmse_nonsecure_entry)) function attribute on page 261
Non-secure function pointer intrinsics on page 353
Overview of building Secure and Non-secure images with the Armv8-M Security Extension

2.3.3 __attribute__((cmse_nonsecure_entry)) function attribute

Declares an entry function that can be called from Non-secure state or Secure state.

Syntax

C code:

```
void __attribute__((cmse_nonsecure_entry)) entry_func(int val)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

C++ code:

```
extern "C" void __attribute__((cmse_nonsecure_entry)) entry_func(int val)
```

Parameters

None.

Operation

Compile Secure code with the maximum capabilities for the target. For example, if you compile with no FPU then the Secure functions do not clear floating-point registers when returning from functions declared as `__attribute__((cmse_nonsecure_entry))`. Therefore, the functions could potentially leak sensitive data.

Example: Using the cmse_nonsecure_entry attribute

```
#include <arm_cmse.h>

void __attribute__((cmse_nonsecure_entry)) entry_func(int val) {
  int state = cmse_nonsecure_caller();

  if (state)
  { // called from non-secure
    // do non-secure work
    ...
  } else
  { // called from within secure
    // do secure work
    ...
  }
}
```

Related information

__attribute__((cmse_nonsecure_call)) function attribute on page 260
Non-secure function pointer intrinsics on page 353
Overview of building Secure and Non-secure images with the Armv8-M Security Extension

2.3.4 __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.

Syntax

```
__attribute__((const))
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

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 attribute is stricter than `__attribute__((pure))` because functions are not permitted to read global memory.

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);
    }
}
```

2.3.5 __attribute__((constructor(priority))) function attribute

This attribute causes the function it is associated with to be called automatically before `main()` is entered.

Syntax

```
__attribute__((constructor(<priority>)))
```

Parameters

`<priority>`

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.

Operation

You can use this attribute for start-up or initialization code.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

In the following example, the constructor functions are called before execution enters `main()`, in the order specified:

```
#include <stdio.h>

void my_constructor1(void) __attribute__((constructor));
void my_constructor2(void) __attribute__((constructor(102)));
void my_constructor3(void) __attribute__((constructor(103)));
void my_constructor1(void) /* This is the 3rd constructor */
{                         /* function to be called */
    printf("Called my_constructor1()\n");
}
void my_constructor2(void) /* This is the 1st constructor */
{                          /* function to be called */
    printf("Called my_constructor2()\n");
}
void my_constructor3(void) /* This is the 2nd constructor */
{                          /* function to be called */
    printf("Called my_constructor3()\n");
}
int main(void)
{
    printf("Called main()\n");
}
```

This example produces the following output:

```
Called my_constructor2()
Called my_constructor3()
Called my_constructor1()
Called main()
```

2.3.6 __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.

Syntax

```
__attribute__((format_arg(<string-index>)))
```

Parameters

`<string-index>`

Specifies the argument that is the format string argument, starting from 1.

Example

The following example declares two functions, `myFormatText1()` and `myFormatText2()`, that provide format strings to `printf()`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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
}
```

```
$ armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -c format_arg_test.c

format_arg_test.c:21:53: warning: format specifies type 'int' but the argument has
 type 'float' [-Wformat]
  printf(myFormatText2("Here is an integer: %d\n"), b); // Type checking. Type
 mismatch results in warning
                                            ~~      ^
                                            %f
1 warning generated.
```

2.3.7 __attribute__((interrupt("type"))) function attribute

This attribute instructs the compiler to generate a function in a manner that is suitable for use as an exception handler.

Syntax

```
__attribute__((interrupt("<type>")))
```

Parameters

`<type>`

Optional, and can be one of the following:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- `IRQ`
- `FIQ`
- `SWI`
- `ABORT`
- `UNDEF`

Restrictions

When using `__attribute__((interrupt("type")))` functions:

- No arguments or return values can be used with the functions.
- The functions are incompatible with `-frwpi`.
- Interrupt `<type>` is case-sensitive.
- `ABORT` can only be a Prefetch Abort. You cannot implement a Data Abort using this attribute.


Note

In Arm®v6-M, Armv7-M, and Armv8-M, the architectural exception handling mechanism preserves registers as follows:

- When pushing context to the stack, the hardware saves eight 32-bit words, comprising xPSR, ReturnAddress, LR (R14), R12, R3, R2, R1, and R0. For more information, see *Exception entry behavior* in the ARMv7-M Architecture Reference Manual.
- A subroutine must preserve the contents of the registers r4-r8, r10, r11, and SP (and r9 in PCS variants that designate r9 as v6). For more information, see *Core registers* in the Procedure Call Standard for the Arm Architecture.

A standard function return can cause an exception return. For some M-profile targets, such as Armv7-M, this attribute generates code that forces the stack to be aligned. This attribute is mandatory on exception handlers.


Note

- For architectures that support A32 and T32 instructions, functions that are specified with this attribute compile to A32 or T32 code depending on whether the compile option specifies A32 code or T32 code.
- For T32 only architectures, for example the Armv6-M architecture, functions specified with this attribute compile to T32 code.
- This attribute is not available for A64 code.
- When targeting the R-profile or A-profile, the type `UNDEF` can only handle **UNDEFINED** A32 instructions and **UNDEFINED** 2-byte sized T32 instructions. Assembly language is required to handle 4-byte sized T32 **UNDEFINED** instructions.

Operation

This attribute affects the code generation of a function as follows:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- If the function is AAPCS, the stack is realigned to 8 bytes on entry.
- For processors that are not based on the M-profile, the attribute preserves all processor registers, rather than only the registers that the AAPCS requires to be preserved. Floating-point registers are not preserved.
- For processors that are not based on the M-profile, the function returns using an instruction that is architecturally defined as a return from exception.

Examples

```
void __attribute__((interrupt)) do_interrupt() { ... }

void __attribute__((interrupt("IRQ"))) do_irq() { ... }
```

2.3.8 __attribute__((malloc)) function attribute

This function attribute indicates that the function can be treated like `malloc` and the compiler can perform the associated optimizations.

Syntax

```
__attribute__((malloc))
```

Parameters

None.

Example

```
void * foo(int b) __attribute__((malloc));
```

2.3.9 __attribute__((naked)) function attribute

This attribute tells the compiler that the function is an embedded assembly function. You can write the body of the function entirely in assembly code using `__asm` statements.

Syntax

```
__attribute__((naked)) <type> <function>;
```

Parameters

None.

Restrictions

The compiler does not generate prologue and epilogue sequences for functions with `__attribute__((naked))`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The compiler supports basic `__asm` statements only in `__attribute__((naked))` functions. Using extended assembly, parameter references, or mixing C code with `__asm` statements might not work reliably.

`armclang` cannot optimize `__attribute__((naked))` functions, because it passes such functions to the integrated assembler to process. However, if the function is not used, `armclang` could remove the whole function.

Examples

```
__attribute__((naked)) int add(int i, int j); // Declaring a function with
 __attribute__((naked)).

__attribute__((naked)) int add(int i, int j)
{
    __asm("ADD r0, r1, #1"); // Basic assembler statements are supported.

/*  Parameter references are not supported inside naked functions: */
/*  __asm (
    "ADD r0, %[input_i], %[input_j]"       // Assembler statement with parameter
 references
    :                                      // Output operand parameter
    : [input_i] "r" (i), [input_j] "r" (j) // Input operand parameter
    );
*/

/*  Mixing C code is not supported inside naked functions: */
/*  int res = 0;
    return res;
*/

}
```

Related information

__asm on page 246

2.3.10 __attribute__((noinline)) function attribute

This attribute suppresses the inlining of a function at the call points of the function.

Syntax

```
<type> <function> __attribute__((noinline));
```

Parameters

None.

Example: Suppress inlining of a function wherever that function is called

```
// Without this attribute, add3() compiles into one 'add 3' instruction
// at a sufficiently high optimization level.
// With this attribute, add3() generates three calls to add1().
int add1(int x) __attribute__((noinline))
{
  return x + 1;
}
int add3(int x)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
{
  return add1(add1(add1(x)));
}
```

Related information

__attribute__((always_inline)) function attribute on page 259
-fno-inline-functions on page 76
Selecting optimization options
Inlining functions

2.3.11 __attribute__((nomerge)) function attribute

This attribute indicates that calls to this function must never be merged during optimization.



Note

This topic describes a [COMMUNITY] feature. See Support level definitions.

Syntax

```
__attribute__((nomerge))
```

Parameters

None.

Operation

The `nomerge` function attribute indicates that calls to the function must never be merged during optimization. For example, it will prevent tail merging otherwise identical code sequences that raise an exception or terminate the program. Tail merging normally reduces the precision of source location information, making stack traces less useful for debugging. This attribute gives greater control over the tradeoff between code size and debug information precision.

2.3.12 __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>, ...)]))
```

Parameters

`[(<arg-index>, ...)]`

Denotes an optional argument index list.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If no argument index list is specified, all pointer arguments are marked as nonnull.


Note

The argument index list is 1-based, rather than 0-based.

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));
```

2.3.13 __attribute__((no_instrument_function)) function attribute

This attribute indicates that calls to this function are not instrumented when compiling using `-finstrument-functions`.


Note

This topic describes a [COMMUNITY] feature. See Support level definitions.

Syntax

```
__attribute__((no_instrument_function))
```

Parameters

None.

Operation

The `no_instrument_function` function attribute indicates that calls to the function must not be instrumented. For example, if you provide your own implementations of the profiling functions `__cyg_profile_func_enter` and `__cyg_profile_func_exit` using the `no_instrument_function` function attribute prevents the profiling functions from being instrumented.

Examples

The following code is an example of how to add counters to count the number of times functions are called and exit:

```
static uint32_t g_enter_counter = 0;
static uint32_t g_exit_counter = 0;

__attribute__((no_instrument_function))
void __cyg_profile_func_enter(void *this_fn,
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
                                void *call_site)
{
    g_enter_counter++;
}

__attribute__((no_instrument_function))
void __cyg_profile_func_exit(void *this_fn,
                             void *call_site)
{
    g_exit_counter++;
}
```

Related information

-finstrument-functions on page 69

2.3.14 __attribute__((no_sanitize("option"))) function attribute

This attribute excludes various sanitizer checks from a particular function.



Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.

Syntax

```
__attribute__((no_sanitize("<option>")))
```

Parameters

`<option>`

One of the following:

`cfi`

Exclude *Control Flow Integrity* (CFI) checks.

`shadow-call-stack`

Exclude the shadow call stack instrumentation from the function, even if enabled globally.

`<ubsan>`

A supported option for *Undefined Behavior Sanitizer* (UBSan), for example, `undefined`. These UBSan options are a [COMMUNITY] feature.

Related information

-fsanitize, -fno-sanitize on page 84
Overview of Control Flow Integrity
Overview of Undefined Behavior Sanitizer

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.3.15 __attribute__((noreturn)) function attribute

This attribute asserts that a function never returns.

Syntax

```
__attribute__((noreturn))
```

Parameters

None.

Restrictions

The return address is not preserved when calling the `noreturn` function. This limits the ability of a debugger to display the call stack.

Operation

Use this attribute to reduce the cost of calling a function that never returns, such as `exit()`. If a `noreturn` function returns to its caller, the behavior is undefined.

2.3.16 __attribute__((not_tail_called)) function attribute

This attribute prevents tail-call optimization on statically bound calls.


Note

This topic describes a [COMMUNITY] feature. See Support level definitions.

Syntax

```
__attribute__((not_tail_called))
```

Parameters

None.

Operation

The `not_tail_called` attribute prevents tail-call optimization on statically bound calls. It has no effect on indirect calls. Virtual functions and functions marked as `always_inline` cannot be marked as `not_tail_called`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.3.17 __attribute__((nothrow)) function attribute

This attribute asserts that a call to a function never results in a C++ exception being sent from the callee to the caller.

Syntax

```
__attribute__((nothrow))
```

Parameters

None.

Operation

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`.

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.

2.3.18 __attribute__((optnone)) function attribute

This attribute disables all optimizations for a function or method, regardless of the optimization level specified on the `armclang` command-line.

Syntax

```
__attribute__((optnone))
```

Parameters

None.

Restrictions

This attribute is incompatible with the `always_inline` attribute.

Operation

This attribute is useful when you need to debug a particular function, but it is not feasible to build the entire application without optimization. Avoiding optimization on the specified function can improve the quality of the debugging information for that function.

Example: Disable all optimizations for a specific function

Create the file `test.c` containing:

```
int test()
{
    return 0;
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
}

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

Compile with:

```
armclang --target=arm-arm-none-eabi -march=armv8-m.main -O3 -S test.c
```

The following assembly code is generated for `main()`:

```
    ...
main:
    ...
    movs    r0, #0
    bx  lr
    ...
```

Change the file `test.c` to:

```
__attribute__((optnone)) int test()
{
    return 0;
}

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

Compile `test.c` again. The following assembly code is generated for `main()` and shows that `test()` is not inlined:

```
    ...
main:
    ...
    push    {r7, lr}
    bl  test
    movs    r0, #0
    pop {r7, pc}
    ...
```

Related information

__attribute__((always_inline)) function attribute on page 259
-O (armclang) on page 203

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.3.19 __attribute__((pcs("calling_convention"))) function attribute

This function attribute specifies the calling convention on targets with hardware floating-point.

Syntax

```
__attribute__((pcs("<calling_convention>")))
```

Parameters

<calling_convention>

One of the following:

aapcs

Use integer registers.

aapcs-vfp

Use floating-point registers.

Restrictions

Valid only for AArch32 state.

Example

```
double foo (float) __attribute__((pcs("aapcs")));
```

2.3.20 __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.

Syntax

```
__attribute__((pure))
```

Parameters

None.

Example

```
int bar(int b) __attribute__((pure));

int bar(int b)
{
    return b++;
}
int foo(int b)
{
    int aLocal=0;
    aLocal += bar(b);
    aLocal += bar(b);
    return 0;
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
}
```

The call to `bar` in this example might be eliminated because its result is not used.

2.3.21 __attribute__((section("name"))) function attribute

The `section` function attribute enables you to place code in different sections of the image.

Syntax

```
<type> <function>(<arguments>) __attribute__((section("<name>")));
```

Parameters

`<name>`

The name you want to use for the section.

Section names must be unique. The compiler produces an error if:

- You use the same section name for different section types.
- You use a section name that is the same as a variable, function, or other symbol in your program.

To place the function at a specific address, use the special name `.ARM.__at_<address>`.

Example: Place a function in a named section

In the following example, the function `foo` is placed into a read-only (RO) section named `new_section` rather than `.text`.

```
int foo(void) __attribute__((section("new_section")));

int foo(void)
{
  return 2;
}
```

Related information

#pragma clang section on page 322
Automatic placement of __at sections on page 664
Relationship between the default armclang-generated sections and scatter-loading input sections on page 592
Language extension compatibility: attributes

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.3.22 __attribute__((target("options"))) function attribute

The `target` function attribute affects the code generation of a function.

Syntax

```
__attribute__((target("<options>")))
```

Parameters

`<options>`

One or more of:

`arm`

Compile the function for A32 state.

`thumb`

Compile the function for T32 state.

`branch-protection=<protection>`

Where `<protection>` is the level or type of protection. One or more of the following values:

`none`

Disables all types of branch protection.

`standard`

Enables *Branch Target Identification* (BTI) and *Pointer Authentication Code* (PAC) branch protection based on the specified `-march=<name>` and `-mcpu=<name>` target options. For more information, see Example: -mbranch-protection=standard.

`bti`

Enables branch protection using BTI. For more information, see Example: -mbranch-protection=bti.

`pac-ret[+<option>+...]`

Enables branch protection using PAC. If you use `pac-ret` protection, you can specify additional options using the `+` separator.

Options for AArch64 state:

`b-key`

Performs pointer authentication using PAC Key B instead of PAC Key A. For more information, see How is the PAC formed?.

`leaf`

Enables pointer authentication for all functions, including leaf functions. A leaf function is a function that does not save the *Link Register* (LR) on the stack.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

pc

Performs pointer authentication using the *Program Counter* (PC) as the second diversifier for return address signing in addition to SP as the first diversifier. A diversifier is also known as a modifier. For more information, see What is the modifier?.

Options for Arm®v8-M targets with the Main Extension:

leaf

Enables pointer authentication for all functions, including leaf functions. A leaf function is a function that does not save the LR on the stack.

Note

You can specify multiple `+<option>` parameters. Include these options immediately after `pac-ret`. For example, `-mbranch-protection=pac-ret+leaf+pc` is valid, but `-mbranch-protection=leaf+pac-ret+pc` is invalid.

For more information, see Example: -mbranch-protection=pac-ret.

Note

This function attribute applies only to functions and takes precedence over the `-mbranch-protection=<protection>` command-line option.

---

harden-pac-ret=<value>

Where `<value>` is one of the following:

none

No return address signing hardening.

load-return-address

Enables return address signing hardening. Must be used with `branch-protection=pac-ret`.

Note

This function attribute applies only to functions and takes precedence over the `-mharden-pac-ret=<option>` command-line option.

Restrictions

The `harden-pac-ret=<value>` function attribute is supported on AArch64 targets only.

Example: Change from A32 to T32 instruction set

Create the file `target_arm_thumb.c` containing the following C code:

```
int arm_function(int a, int b)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
{
    return a + b % a;
}

__attribute__((target("thumb"))) int thumb_function(int a, int b)
{
    return a + b % a;
}
```

Compile with:

```
armclang --target=arm-arm-none-eabi -march=armv7-r -marm -O1 -S target_arm_thumb.c
```

The following assembly is generated using the `.code` directive:

```
        ...
        .eabi_attribute 8, 1    @ Tag_ARM_ISA_use
        .eabi_attribute 9, 2    @ Tag_THUMB_ISA_use
        ...
        .code   32
arm_function:
        ...
        .code   16
thumb_function:
        ...
```

Example: Change from T32 to A32 instruction set

Create the file `target_thumb_arm.c` containing the following C code:

```
__attribute__((target("arm"))) int arm_function(int a, int b)
{
    return a + b % a;
}

int thumb_function(int a, int b)
{
    return a + b % a;
}
```

Compile with:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -mthumb -O1 -S target_arm_thumb.c
```

The following assembly is generated using the `.code` directive:

```
        ...
        .eabi_attribute 8, 1    @ Tag_ARM_ISA_use
        .eabi_attribute 9, 2    @ Tag_THUMB_ISA_use
        ...
        .code   32
arm_function:
        ...
        .code   16
thumb_function:
        ...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: bti+pac-ret+leaf

```
void __attribute__((target("branch-protection=bti+pac-ret+leaf"))) foo() { ... }
```

Example: Return address signing hardening in Armv8.3-A

Create the file `address.c` containing the following C code:

```
extern void bar();

#ifdef ADD_ATTRIBUTE
__attribute__((target("harden-pac-ret=load-return-address,branch-protection=pac-
ret")))
#endif
int foo(int cond)
{
    bar();
    if (cond)
    {
        return 0;
    }
    return 1;
}
```

Compile `address.c` with:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.3-a -O2 -S -o address.s
 address.c
```

The generated file `address.s` contains:

```
    ...
foo:
    stp x30, x19, [sp, #-16]!           // 16-byte Folded Spill
    mov w19, w0
    bl  bar
    cmp w19, #0
    cset  w0, eq
    ldp x30, x19, [sp], #16             // 16-byte Folded Reload
    ret
    ...
```

Compile again with the return address signing option `-mbranch-protection=pac-ret` but without return address signing hardening:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.3-a -O2 -mbranch-
protection=pac-ret -S -o address_ra_sign.s address.c
```

The generated assembly now contains:

```
    ...
foo:
    paciasp
    stp x30, x19, [sp, #-16]!           // 16-byte Folded Spill
    mov w19, w0
    bl  bar
    cmp w19, #0
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    cset  w0, eq
    ldp x30, x19, [sp], #16             // 16-byte Folded Reload
    retaa
    ...
```

Compile again with the `__attribute__((target("harden-pac-ret=load-return-address,branch-protection=pac-ret")))` function attribute by defining the preprocessor macro `ADD_ATTRIBUTE`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.3-a -O2 -DADD_ATTRIBUTE -S -o
 address_attr.s address.c
```

The return address signing command-line option `-mbranch-protection=pac-ret` has no effect, so the option must be included as part of the function attribute.

The generated assembly now contains:

```
    ...
foo:
    paciasp
    stp x30, x19, [sp, #-16]!           // 16-byte Folded Spill
    mov w19, w0
    bl  bar
    cmp w19, #0
    cset    w0, eq
    ldp x30, x19, [sp], #16             // 16-byte Folded Reload
    autiasp
    mov x16, x30
    xpaci   x16
    ldr w16, [x16]
    ret
    ...
```

The `AUTIASP` and `RET` code sequence here contains an `XPACI` instruction. Therefore, the code is not vulnerable to PACMAN.

Example: branch-protection=pac-ret is required with harden-pac-ret=load-return-address

Create the file `address.c` containing the following C code:

```
extern void bar();

__attribute__((target("harden-pac-ret=load-return-address"))) int foo(int cond)
{
    bar();
    if (cond)
    {
        return 0;
    }
    return 1;
}
```

Compile `address.c` with the return address signing option `-mbranch-protection=pac-ret`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.3-a -O2 -mbranch-
protection=pac-ret -S -o address.s address.c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The compiler generates the following warning:

```
address.c:3:23: warning: 'harden-pac-ret' attribute requires 'branch-protection=pac-
ret'; 'target' attribute ignored
      [-Wignored-attributes]
    3 | __attribute__((target("harden-pac-ret=load-return-address"))) int foo(int
 cond)
      |                       ^
1 warning generated.
```

When using the function attibute with `harden-pac-ret=load-return-address`, you must also add `branch-protection=pac-ret` as part of the attribute, and not the command-line option. Using the command-line option `-mbranch-protection=<protection>` has no effect.

Related information

-marm on page 140
-mbranch-protection on page 144
-mharden-pac-ret on page 176
-mthumb on page 193
PACMAN: Attacking ARM Pointer Authentication with Speculative Execution paper

2.3.23 __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.

Syntax

```
__attribute__((unused))
```

Parameters

None.

Operation

By default, the compiler does not warn about unused functions. Use `-Wunused-function` to enable this warning specifically, or use an encompassing `-W` value such as `-Wall`.

The `__attribute__((unused))` attribute can be useful if you usually want to warn about unused functions, but want to suppress warnings for a specific set of functions.

Example

```
static int unused_no_warning(int b) __attribute__((unused));

static int unused_no_warning(int b)
{
  return b++;
}

static int unused_with_warning(int b);
static int unused_with_warning(int b)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
{
  return b++;
}
```

Compiling this example with `-Wall` results in the following warning:

```
armclang --target=aarch64-arm-none-eabi -c test.c -Wall

test.c:10:12: warning: unused function 'unused_with_warning' [-Wunused-function]

static int unused_with_warning(int b)
           ^
1 warning generated.
```

Related information

__attribute__((unused)) variable attribute on page 300

2.3.24 __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.

Syntax

```
__attribute__((used))
```

Parameters

None.

Operation

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, by using `__attribute__((used))`.

Example

```
static int lose_this(int);
static int keep_this(int) __attribute__((used));     // retained in object file
static int keep_this (int arg) {
  return (arg+1);
}
static int keep_this_too(int) __attribute__((used)); // retained in object file
static int keep_this_too (int arg) {
  return (arg-1);
}

int main (void) {
  for (;;);
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
}
```

Related information

Elimination of unused sections on page 611

2.3.25 __attribute__((value_in_regs)) function attribute

The `value_in_regs` function attribute is compatible with functions whose return type is a structure. It changes the calling convention of a function so that the returned structure is stored in the argument registers rather than being written to memory using an implicit pointer argument.



Note

When using `__attribute__((value_in_regs))`, the calling convention only uses integer registers.

Syntax

```
__attribute__((value_in_regs)) return-type function-name([argument-list]);
```

Parameters

`return-type`

The type of the returned structure that conforms to certain restrictions as described in *Restrictions*.

Restrictions

When targeting AArch32, the returned structure can be up to 16 bytes to fit in four 32-bit argument registers. When targeting AArch64, the returned structure can be up to 64 bytes to fit in eight 64-bit argument registers. If the structure returned by a function that is qualified by `__attribute__((value_in_regs))` is too large, the compiler generates an error.

Each field of the returned structure must occupy exactly one or two integer registers, and must not require implicit padding of the structure. Anything else, including bitfields, is incompatible.

Nested structures are allowed with the same restriction that the nested structure as a whole and its individual members must occupy exactly one or two integer registers.

Unions are allowed if they have at least one maximal-size member that occupies exactly one or two integer registers. The other fields within the union can have any field type.

The allowed field types are:

- `signed int` (AArch32 only).
- `unsigned int` (AArch32 only).
- `signed long`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- `unsigned long`.
- `signed long long`.
- `unsigned long long`.
- `pointer`.
- `structure` containing any of the types in this list.
- `union` whose maximal-size member is any of the types in this list.

If the structure type returned by a function that is qualified by `__attribute__((value_in_regs))` violates any of the preceding rules, then the compiler generates the corresponding error.

If a virtual function declared as `__attribute__((value_in_regs))` is to be overridden, the overriding function must also be declared as `__attribute__((value_in_regs))`. If the functions do not match, the compiler generates an error.

A function that is declared as `__attribute__((value_in_regs))` is not function-pointer-compatible with a normal function of the same type signature. If a pointer to a function that is declared as `__attribute__((value_in_regs))` is initialized with a pointer to a function that is not declared as `__attribute__((value_in_regs))`, then the compiler generates a warning.

The return type of a function that is declared as `__attribute__((value_in_regs))` must be known at the point of the function declaration. If the return type is an incomplete type, the compiler generates a corresponding error.

Operation

Declaring a function `__attribute__((value_in_regs))` can be useful when calling functions that return more than one result.

Example

```
struct ReturnType
{
    long a;
    void *ptr;
    union U
    {
       char c;
       short s;
       int i;
       float f;
       double d;
       struct S1 {long long ll;} s1;
    } u;
};

extern __attribute__((value_in_regs)) struct ReturnType g(long y);
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.3.26 __attribute__((visibility("visibility_type"))) function attribute

This function attribute affects the visibility of ELF symbols.

Default

If you do not specify visibility, then the default type is `default` for `extern` declarations and `hidden` for everything else.

Syntax

```
__attribute__((visibility("<visibility_type>")))
```

Parameters

`<visibility_type>`

One of the following:

`default`

The symbol is public and corresponds to external linkage. You can export a function regardless of the value specfied with `-fvisibility`.

`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.

`protected`

The symbol is placed into the dynamic symbol table, but references within the defining module bind to the local symbol. That is, another module cannot override the symbol.

Operation

This attribute overrides other settings that determine the visibility of symbols.

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.

In the case of namespace declarations, the visibility attribute applies to all function and variable definitions.

Example

```
void __attribute__((visibility("protected"))) foo()
{
   ...
}
```

Related information

-fvisibility on page 110
__attribute__((visibility("visibility_type"))) variable attribute on page 302

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.3.27 __attribute__((weak)) function attribute

Functions defined with `__attribute__((weak))` export their symbols weakly.

Syntax

```
__attribute__((weak)) <type> <function>(<argument>);
```

Parameters

None.

Operation

Functions declared with `__attribute__((weak))` and then defined without `__attribute__((weak))` behave as weak functions.

Example

```
__attribute__((weak)) int Function_Attributes_weak_0(int b);
```

Related information

__attribute__((weak)) variable attribute on page 303
Weak references and definitions on page 603

2.3.28 __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>")))
```

Parameters

`<target>`

The target symbol.

Restrictions

This attribute can only be used on functions with static linkage.

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();
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
  ...
}
```

2.3.29 __attribute__((aligned)) type attribute

The `aligned` type attribute specifies a minimum alignment for the type. The `aligned` type attribute only increases the alignment of a `struct` or `struct` member, and does not decrease it.

Default

When an `aligned` value is not specified, the type aligns to the default alignment:

- For AArch32, the default alignment is 8 bytes.
- For AArch64, the default alignment is 16 bytes.

Syntax

```
<type> __attribute__((aligned));
<type> __attribute__((aligned(<size>)));
```

Parameters

`<size>`

The minimum alignment for the type, in bytes.

Operation

You can use the `packed` and `aligned` attributes together on the same member to set the alignment to a value. The value is less than the default value, but greater than one.

When used as part of a typedef, the `aligned` type attribute can both increase and decrease alignment.

You can also use the `aligned` variable attribute to both increase and decrease the alignment of a variable. For more information, see __attribute__((aligned)) variable attribute.

Example: Alignment of a packed 64-bit unsigned integer

Create `alignment.c` containing the following code:

```
#include <stdint.h>
#include <stdio.h>
#include <stddef.h>

struct ints {
    uint8_t chA;
    uint64_t dwB __attribute__((packed)) __attribute__((aligned(4)));
};

int main() {
  printf(
      "ints is size %d align %d, with chA at offset %d and dwB at offset %d\n",
      sizeof(struct ints), _Alignof(struct ints), offsetof(struct ints, chA),
      offsetof(struct ints, dwB));
  return 0;
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
}
```

Compile `alignment.c` using the following command:

```
armclang --target=arm-arm-none-eabi -march=armv7-a alignment.c -o alignment.axf
```

Run the `alignment.axf` image on a suitable target. The output is:

```
ints is size 12 align 4, with chA at offset 0 and dwB at offset 4
```

Example: Other __attribute__((aligned)) type attribute examples

More examples of the `__attribute__((aligned))` type attribute are available in __attribute__((aligned)) type attribute examples.

Related information

__attribute__((packed)) type attribute on page 289
__attribute__((aligned)) variable attribute on page 292
Alignment support in Arm Compiler for Embedded 6

2.3.30 __attribute__((packed)) type attribute

The `packed` type attribute specifies that a type must have the smallest possible alignment. This attribute only applies to `struct` and `union` types.

Syntax

```
struct __attribute__((packed)) <structure_name> { ... };
```

Parameters

None.

Operation

You must access a `packed` member of a `struct` or `union` directly from a variable of the containing type. Taking the address of such a member produces a normal pointer that might be unaligned. The compiler assumes that the pointer is aligned. Dereferencing such a pointer can be unsafe even when the target supports unaligned accesses, because certain instructions always require word-aligned addresses.


Note

If you take the address of a packed member, the compiler usually generates a warning.

When you specify `__attribute__((packed))` to a structure or union, it applies to all members of the structure or union. If a packed structure has a member that is also a structure, then this

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

member structure has an alignment of 1-byte. However, the packed attribute does not apply to the members of the member structure. The members of the member structure continue to have their natural alignment.

Example: Correct and incorrect use of a packed structure

```
struct __attribute__((packed)) my_struct_packed
{
  char x;
  short y;
};

short get_y(struct my_struct_packed *s)
{
    // Correct usage: the compiler does not use unaligned accesses
    // unless they are allowed.
    return s->y;
}

short get2_y(struct my_struct_packed *s)
{
    short *p = &s->y; // Incorrect usage: 'p' might be an unaligned pointer.
    return *p;        // This might cause an unaligned access.
}
```

Related information

__attribute__((aligned)) type attribute on page 288
__attribute__((aligned)) variable attribute on page 292
-munaligned-access, -mno-unaligned-access on page 197
Alignment support in Arm Compiler for Embedded 6

2.3.31 __attribute__((transparent_union)) type attribute

The `transparent_union` type attribute enables you to specify a transparent union type.

Syntax

```
__attribute__((transparent_union))
```

Parameters

None.

Operation

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.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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 */
}
```

2.3.32 __attribute__((alias)) variable attribute

This variable attribute enables you to specify multiple aliases for a variable.

Syntax

```
<type> <newname>  __attribute__((alias("<oldname>")));
```

Parameters

<oldname>

The name of the variable to be aliased.

<newname>

The new name of the aliased variable.

Operation

Aliases must be declared in the same translation unit as the definition of the original variable.


Note

Aliases cannot be specified 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.


Note

Function names can also be aliased using the corresponding function attribute `__attribute__((alias))`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Create an alias for a variable

Create the file `alias.c` containing the following C code:

```
#include <stdio.h>

int oldname = 1;
extern int newname __attribute__((alias("oldname"))); // declaration
void print_alias(void){
    printf("newname = %d\n", newname); // prints 1
}
```

Compile with:

```
armclang --target=arm-arm-none-eabi -march=armv8a -S alias.c
```

This code compiles to:

```
    ...
    .type     oldname,%object
    .section  .data.oldname,"aw",%progbits
    .globl    oldname
    .p2align  2, 0x0
oldname:
    .long     1
    .size     oldname, 4

    .type     .L.str,%object
    .section  .rodata.str1.1,"aMS",%progbits,1
.L.str:
    .asciz    "newname = %d\n"
    .size     .L.str, 14

    .globl    newname
    .hidden   newname
.set newname, oldname
    ...
```

2.3.33 __attribute__((aligned)) variable attribute

The `aligned` variable attribute specifies a minimum alignment for a variable or structure field, measured in bytes. The `aligned` variable attribute only increases the alignment of a `struct` or `struct` member, and does not decrease it.

Default

When no value is given, the alignment used is the maximum alignment for a scalar data type:

- For AArch32, the default alignment is 8 bytes.
- For AArch64, the default alignment is 16 bytes.

Syntax

```
__attribute__((aligned))
__attribute__((aligned(<size>)))
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<size>

The alignment size in bytes. It must be a power of 2.

Operation

For a variable that is not in a structure, the minimum alignment is the natural alignment of the variable type. To set the alignment in a structure to any value greater than 0, use the `__attribute__((packed))` variable attribute. Without `__attribute__((packed))`, the minimum alignment is the natural alignment of the variable type.

Example: Align a variable on a 16-byte boundary

```
int x __attribute__((aligned(16)));
```

Example: Maximum alignment for a scalar data type

```
// When no alignment value is given, the alignment used is the maximum
// alignment for a scalar data type.
//   For A32, the maximum is 8 bytes.
//   For A64, the maximum is 16 bytes.
short my_array[3] __attribute__((aligned));
```

Example: Attempt to reduce alignment below the natural alignment of a data type

```
// Cannot decrease the alignment below the natural alignment of the type.
// Aligns on a 4-byte boundary.
int my_array_reduced[3] __attribute__((aligned(2)));
```

Example: Effect of decreasing alignment below the natural alignment

Create the file `alignment.c` containing the following code:

```
#include <stdint.h>
#include <stdio.h>
#include <stddef.h>

#define STR(s) #s

// Aligns on a 16-byte boundary
int x __attribute__((aligned(16)));

// When no value is given, the alignment used is the maximum alignment
// for a scalar data type:
//   For A32, the maximum is 8 bytes.
//   For A64, the maximum is 16 bytes.
short my_array[3] __attribute__((aligned));

// Cannot decrease the alignment below the natural alignment of the type.
// Aligns on a 4-byte boundary.
int my_array_reduced[3] __attribute__((aligned(2)));

// b aligns on an 8-byte boundary for A32 and a 16-byte boundary for A64
struct my_struct
{
    char a;
    int b __attribute__((aligned));
};

// 'aligned' on a struct member cannot decrease the alignment below the
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
// natural alignment of that member. b aligns on a 4-byte boundary.
struct my_struct_reduced
{
    char a;
    int b __attribute__((aligned(2)));
};

// Combine 'packed' and 'aligned' on a struct member to set the alignment for
// that member to any value. b aligns on a 2-byte boundary.
struct my_struct_packed
{
    char a;
    int b __attribute__((packed)) __attribute__((aligned(2)));
};

int main() {
#define SHOW_STRUCT(t)                                                  \
  do {                                                                  \
    printf(STR(t) " is size %zd, align %zd\n", sizeof(struct t),     \
           _Alignof(struct t));                                         \
    printf("  a is at offset %zd\n", offsetof(struct t, a));         \
    printf("  b is at offset %zd\n", offsetof(struct t, b));         \
  } while (0)

  SHOW_STRUCT(my_struct);
  SHOW_STRUCT(my_struct_reduced);
  SHOW_STRUCT(my_struct_packed);
  return 0;
}
```

To compile `alignment.c`, enter:

```
armclang --target=arm-arm-none-eabi -march=armv7-a alignment.c -o alignment.axf
```

Run the `alignment.axf` image on a suitable target. The output is:

```
my_struct is size 16, align 8
  a is at offset 0
  b is at offset 8
my_struct_reduced is size 8, align 4
  a is at offset 0
  b is at offset 4
my_struct_packed is size 6, align 2
  a is at offset 0
  b is at offset 2
```

Example: Alignment of variables

Create the file `test_scalar.c` containing the following code:

```
#include <stdint.h>
#include <stdio.h>
#include <stddef.h>

#define STR(s) #s

// 'aligned' can both increase and decrease the alignment of a variable.
int a;
int b __attribute__((aligned(2)));
int c __attribute__((aligned(8)));
// when an 'aligned' value is not specified, the variable aligns to the default
 alignment for AArch32, that is, 8 bytes (the default alignment for AArch64 is 16
 bytes).
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
int d __attribute__((aligned));

void test_scalar() {
#define SHOW_VAR(v) \
  do { printf("int " STR(v) " is size %zd, align %zd\n", sizeof(v), _Alignof(v)); }
 while (0)
  SHOW_VAR(a);
  SHOW_VAR(b);
  SHOW_VAR(c);
  SHOW_VAR(d);
}

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

Compile `test_scalar.c` using the following command:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -Wno-gnu-alignof-expression
 test_scalar.c -o test_scalar.axf -Wall
```

Run the `test_scalar.axf` image. The output is:

```
int a is size 4, align 4
int b is size 4, align 2
int c is size 4, align 8
int d is size 4, align 8
```

Related information

__attribute__((aligned)) type attribute on page 288
__attribute__((packed)) variable attribute on page 296
Alignment support in Arm Compiler for Embedded 6

2.3.34 __attribute__((common)), __attribute__((nocommon)) variable attribute

Generates common zero-initialized values for tentative definitions. Tentative definitions are declarations of variables with no storage class and no initializer.

Syntax

```
__attribute__((common))
__attribute__((nocommon))
```

Parameters

None.

Operation

The `common` attribute places the tentative definitions in a common block.

The `nocommon` attribute generates individual zero-initialized definitions for tentative definitions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

These attributes override the `-fno-common` and `-fcommon` command-line options.

Related information

-fcommon, -fno-common on page 52

2.3.35 __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.

Syntax

```
__attribute__((deprecated))
```

Parameters

None.

Operation

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.

Example

```
extern int deprecated_var __attribute__((deprecated));

void foo()
{
    deprecated_var=1;
}
```

Compiling this example generates a warning:

```
armclang --target=aarch64-arm-none-eabi -c test_deprecated.c

test_deprecated.c:4:3: warning: 'deprecated_var' is deprecated [-Wdeprecated-
declarations]

  deprecated_var=1;
  ^
test_deprecated.c:1:12: note: 'deprecated_var' has been explicitly marked deprecated
 here
  extern int deprecated_var __attribute__((deprecated));
  ^
1 warning generated.
```

2.3.36 __attribute__((packed)) variable attribute

You can specify the `packed` variable attribute on fields that are members of a structure or union. The attribute specifies that a member field has the smallest possible alignment. That is, one byte

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

for a variable field, and one bit for a bitfield, unless you specify a larger value with the `aligned` attribute.

Syntax

```
<type> <variable> __attribute__((packed));
```

Parameters

None.

Operation

You must access a `packed` member of a structure or union directly from a variable of the structure or union. Taking the address of such a member produces a normal pointer that might be unaligned. The compiler assumes that the pointer is aligned. Dereferencing such a pointer can be unsafe even when the target supports unaligned accesses, because certain instructions always require word-aligned addresses.



Note

In most cases, if you take the address of a packed member, the compiler generates a warning.

Example: Packed member of a structure

```
struct
{
    char a;
    int b __attribute__((packed));
} Variable_Attributes_packed_0;
```

Related information

__attribute__((packed)) type attribute on page 289
__attribute__((aligned)) variable attribute on page 292
Alignment support in Arm Compiler for Embedded 6

2.3.37 __attribute__((section("name"))) variable attribute

The `section` attribute specifies that a variable must be placed in a particular data section.

Syntax

```
<type> <variable> __attribute__((section("<name>")));
```

Parameters

`<name>`

The name you want to use for the section.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Section names must be unique. The compiler produces an error if:

- You use the same section name for different section types.
- You use a section name that is the same as a variable, function, or other symbol in your program.

To place ZI data in a named section, the `<name>` must start with the `.bss.` prefix. Non-ZI data cannot be placed in a section name with the `.bss.` prefix.

To place the variable at a specific address, use the special name:

`.ARM.__at_<address>`
To place non-ZI data.

`.bss.ARM.__at_<address>`
To place ZI data.

Operation

Normally, `armclang` places the data 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, writable variables are placed in RW data sections.

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 */
int my_zi __attribute__((section(".bss.my_zi_section")));
```

Related information

#pragma clang section on page 322
Manual placement of __at sections on page 666
Language extension compatibility: attributes

2.3.38 __attribute__((tls_model("model"))) variable attribute

The `tls_model` variable attribute sets the *Thread Local Storage* (TLS) model of a particular `__thread` variable, overriding the `-ftls-model` command-line option on a per-variable basis.

Syntax

```
__attribute__((tls_model("<model>")))
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<model>

One of the following, ordered from most restrictive to least restrictive:

local-exec

Handles access to symbols that exist within the executable or module at static link time, where all offsets can be resolved in relation to the thread pointer.

initial-exec

Handles access to symbols from external modules that are initially loaded, where all offsets are run-time constants.

global-dynamic

The most generic model, which handles external access to dynamically loaded modules. This model is sometimes referred to as the *general dynamic model* in TLS specification and ABI documents.


Note

A fourth model, `local-dynamic`, is not supported. If you use `local-dynamic`, then access is handled in the same way as `global-dynamic`.

Operation

The `__thread` keyword and `thread_local` keyword are equivalent. The `__thread` keyword is universally accepted as a GNU extension in both C and C++. The `thread_local` keyword is supported for C++ with `-std=c++11` or later. The `thread_local` keyword is only supported for C in Arm® Compiler for Embedded version 6.19 and later when used with the `-std=c2x` [COMMUNITY] feature for C23 support.

The compiler attempts to select the appropriate TLS model for each `__thread` variable as follows:

- If `-fpic` is used, then the compiler makes the assumption that the code being compiled can be used in a shared object. Therefore, the `general-dynamic` model is used.
- If `-fpic` is not used, then the compiler makes the assumption that the code being compiled is for a statically linked executable. Either `local-exec` or `initial-exec` is used.
  - `local-exec` is used for `__thread` variables that are defined and used within the same source file.
  - `initial-exec` is used for `__thread` variables that are defined using `extern`.

In other words the compiler selects the least flexible (and most efficient) TLS model which is guaranteed to work in each case.

You can override these choices using either the `-ftls-model` command-line option or `tls_model` variable attribute if you know it is safe to do so.

Example: Assign the local-exec model to the function test()

```
void __attribute__((tls_model("local-exec"))) test() { ... }
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: AArch64 TLS local-exec static linking example

For an example of AArch64 TLS `local-exec` static linking with source code that you can build and run, see AArch64 TLS local-exec static linking example.

Related information

-ftls-model on page 105
Thread Local Storage

2.3.39 __attribute__((uninitialized)) variable attribute

Requests that a trivial automatic variable is uninitialized.

Syntax

```
<type> <variable> __attribute__((uninitialized));
```

Parameters

None.

Operation

Use the `__attribute__((uninitialized))` variable attribute to ensure individual automatic variables are not initialized when using the `-ftrivial-auto-var-init=pattern` or `-ftrivial-auto-var-init=zero` options.

For example, you might want to disable the initialization of large stack arrays, because initializing them might be time-consuming.

Example

```
int not_initialized __attribute__((uninitialized));
```

Related information

-ftrivial-auto-var-init on page 108
Automatic variable initialization

2.3.40 __attribute__((unused)) variable attribute

The compiler can warn if a variable is declared but is never referenced. The __attribute__((unused)) attribute informs the compiler to expect an unused variable, and tells it not to issue a warning.

Syntax

```
__attribute__((unused))
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

By default, the compiler does not warn about unused variables. Use `-Wunused-variable` to enable this warning specifically, or use an encompassing `-W` value such as `-Weverything`.

The `__attribute__((unused))` attribute can be used to warn about most unused variables, but suppress warnings for a specific set of variables.

Example

```
void foo()
{
    static int aStatic =0;
    int aUnused __attribute__((unused));
    int bUnused;
    aStatic++;
}
```

When compiled with a suitable `-W` setting, the compiler warns that `bUnused` is declared but never referenced, but does not warn about `aUnused`:

```
armclang --target=aarch64-arm-none-eabi -c test_unused.c -Wall

test_unused.c:5:7: warning: unused variable 'bUnused' [-Wunused-variable]

  int bUnused;
      ^
1 warning generated.
```

Related information

__attribute__((unused)) function attribute on page 282

2.3.41 __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.

Syntax

```
__attribute__((used))
```

Parameters

None.

Operation

Data marked with `__attribute__((used))` is tagged in the object file to avoid removal by linker unused section removal.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

Static functions can also be marked as used, by using `__attribute__((used))`.

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 information

Elimination of unused sections on page 611

2.3.42 __attribute__((visibility("visibility_type"))) variable attribute

This variable attribute affects the visibility of ELF symbols.

Default

If you do not specify visibility, then the default type is `default` for `extern` declarations and `hidden` for everything else.

Syntax

```
__attribute__((visibility("<visibility_type>")))
```

Parameters

`<visibility_type>`

One of the following:

`default`

The symbol is public and corresponds to external linkage. You can export a variable regardless of the value specfied with `-fvisibility`.

`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.

`protected`

The symbol is placed into the dynamic symbol table, but references within the defining module bind to the local symbol. That is, another module cannot override the symbol.

Operation

This attribute overrides other settings that determine the visibility of symbols.

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.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

In the case of namespace declarations, the visibility attribute applies to all function and variable definitions.

Example

```
int __attribute__((visibility("hidden"))) foo = 1;   // hidden in object file
```

Related information

-fvisibility on page 110
__attribute__((visibility("visibility_type"))) function attribute on page 285

2.3.43 __attribute__((weak)) variable attribute

Generates a weak symbol for a variable, rather than the default symbol.

Syntax

```
__attribute__((weak)) <type> <variable>;
```

Parameters

None.

Operation

At link time, strong symbols override weak symbols. This attribute replaces a weak symbol with a strong symbol, by choosing a particular combination of object files to link.

Example

```
__attribute__((weak)) int foo;
```

Related information

__attribute__((weak)) function attribute on page 286
Weak references and definitions on page 603

2.3.44 __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.

Syntax

```
__attribute__((weakref("<target>")))
```

Parameters

<target>

The target symbol.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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;
  ...
}
```

2.3.45 __attribute__((aligned)) type attribute examples

These examples show how to use the `__attribute__((aligned))` type attribute.

For more information about the `__attribute__((aligned))` type attribute, see __attribute__((aligned)) type attribute.

Example: Alignment of structures

Create the file `test_struct_1.c` containing the following code:

```
#include <stdint.h>
#include <stdio.h>
#include <stddef.h>

#define STR(s) #s

// The size of x is 1 byte, the size of y is 4 bytes.
struct S {
  char x;
  int y;
};

struct T {
  char x;
  // 'aligned' on a structure member increases its alignment.
  int y __attribute__((aligned(8)));
};

struct U {
  char x;
  // 'aligned' on a structure member cannot decrease its alignment.
  // y still has alignment 4. Structure U is identical to structure S.
  int y __attribute__((aligned(2)));
};

struct V {
  char x;
  // 'packed' reduces a structure member alignment to 1.
  int y __attribute__((packed));
};

struct W {
  char x;
  // Combine 'packed' and 'aligned' on a structure member to set the
  // alignment of that member to any value.
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
  int y __attribute__((packed)) __attribute__((aligned(2)));
};

// 'packed' on a structure applies to all of its members.
// Structure X is identical to structure S.
struct X {
  char x;
  int y;
} __attribute__((packed));

// 'aligned' on a structure applies to the structure as a whole,
// not individual members, so does not affect layout of members.
struct Y {
  char x;
  int y;
} __attribute__((aligned(8)));

// 'aligned' cannot decrease the alignment. Y2 still has alignment 4.
// Structure Y2 is identical to structure S.
struct Y2 {
  char x;
  int y;
} __attribute__((aligned(2)));

// typedef creates the type alias named Y3. The Y3 type is the same as
// structure S but with an alignment of 2 bytes. If you use the aligned
// attribute on a typedef rather than on a structure, it only changes the
// alignment of the type and does not affect the layout of the members.
typedef struct S __attribute__((aligned(2))) Y3;

// 'packed' on a structure applies to all of its members.
// 'aligned' on an individual member increases the alignment.
struct Z {
  char x;
  int y __attribute__((aligned(2)));
} __attribute__((packed));

void test_struct_1() {
#define SHOW_STRUCT(t)                                                  \
  do {                                                                  \
    printf(STR(t) " is size %zd, align %zd\n", sizeof(struct t),        \
           _Alignof(struct t));                                        \
    printf("  x is at offset %zd\n", offsetof(struct t, x));            \
    printf("  y is at offset %zd\n", offsetof(struct t, y));            \
  } while (0)

  SHOW_STRUCT(S);
  SHOW_STRUCT(T);
  SHOW_STRUCT(U);
  SHOW_STRUCT(V);
  SHOW_STRUCT(W);
  SHOW_STRUCT(X);
  SHOW_STRUCT(Y);
  SHOW_STRUCT(Y2);
  printf("Y3 is size %zd, align %zd\n", sizeof(Y3), _Alignof(Y3));
  printf("  x is at offset %zd\n", offsetof(Y3, x));
  printf("  y is at offset %zd\n", offsetof(Y3, y));
  SHOW_STRUCT(Z);
}

int main() {
  test_struct_1();
  return 0;
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Compile `test_struct_1.c` using the following command:

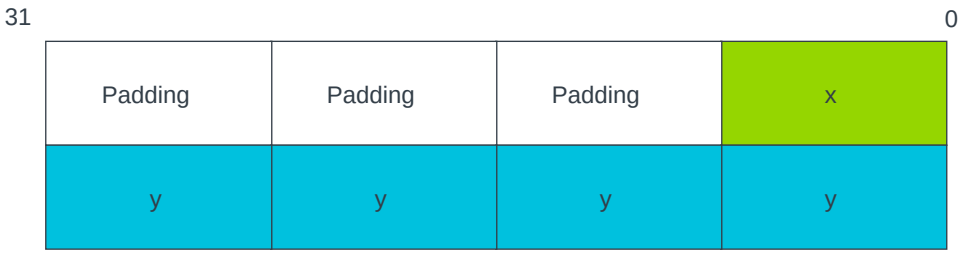
```
armclang --target=arm-arm-none-eabi -march=armv7-a -Wno-gnu-alignof-expression
 test_struct_1.c -o test_struct_1.axf -Wall
```

Run the `test_struct_1.axf` image. The output is:

The minimum alignment of int y in structure S is the natural alignment of the int y variable type

```
S is size 8, align 4
  x is at offset 0
  y is at offset 4
```

Figure 2-1: Minimum alignment of int y in structure S is the natural alignment of the int y variable type

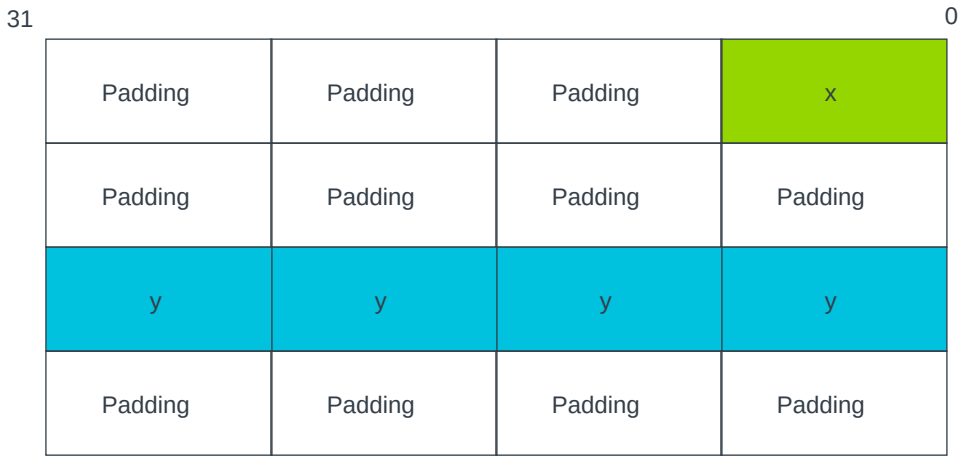


The alignment of int y in structure T increases from the natural alignment of 4 bytes to 8 bytes

```
T is size 16, align 8
  x is at offset 0
  y is at offset 8
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-2: Alignment of int y in structure T increases from the natural alignment of 4 bytes to 8 bytes



Structure U is identical to structure S

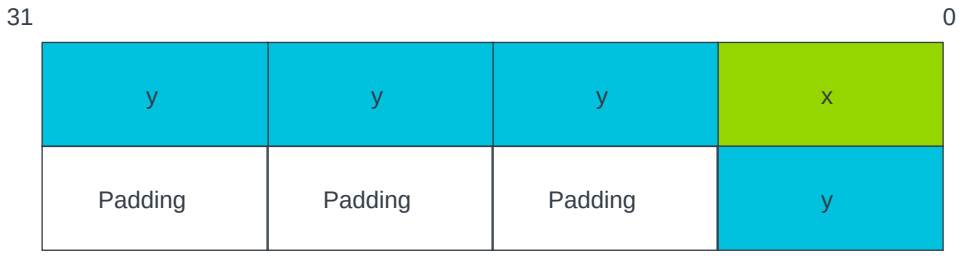
```
U is size 8, align 4
  x is at offset 0
  y is at offset 4
```

The packed type attribute minimizes the memory required for x and y in structure V

```
V is size 5, align 1
  x is at offset 0
  y is at offset 1
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

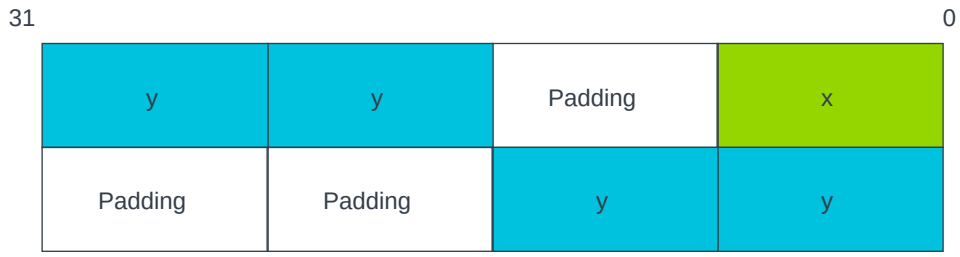
Figure 2-3: Packed type attribute minimizes the memory required for x and y in structure V



Following the packed type attribute with the `aligned(2)` type attribute sets the minimum alignment to 2 bytes

```
W is size 6, align 2
  x is at offset 0
  y is at offset 2
```

Figure 2-4: Packed type attribute with the aligned(2) type attribute sets the minimum alignment to 2 bytes



Structure X is identical to structure V

```
X is size 5, align 1
  x is at offset 0
  y is at offset 1
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Structure Y is similar to structure S. The alignment of structure Y is 8 bytes compared to the alignment of 4 bytes for structure S, but this does not affect the layout.

```
Y is size 8, align 8
  x is at offset 0
  y is at offset 4
```

Structure Y2 is identical to structure S

```
Y2 is size 8, align 4
  x is at offset 0
  y is at offset 4
```

Structure Y3 is similar to structure S. The alignment of structure Y3 is 2 bytes compared to the alignment of 4 bytes for structure S, but this does not affect the layout.

```
Y3 is size 8, align 2
  x is at offset 0
  y is at offset 4
```

Structure Z is identical to structure W

```
Z is size 6, align 2
  x is at offset 0
  y is at offset 2
```

Example: Alignment of structures containing another structure

Create the file `test_struct_2.c` containing the following code:

```
#include <stdint.h>
#include <stdio.h>
#include <stddef.h>

#define STR(s) #s

int d __attribute__((aligned));

struct S {
  char x;
  int y;
};

// 'packed' and 'aligned' only affect the layout of the structure they are
// directly applied to, not any structure types they contain.
struct A {
  struct S z __attribute__((aligned(2)));
} __attribute__((packed));

// 'aligned' applies to the member, so it increases the alignment of
// z back up to 2 after the 'packed' reduced it to 1.
struct B {
  struct {
    char x;
    int y;
  } z __attribute__((aligned(2)));
} __attribute__((packed));
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
// 'aligned' applies to the inner structure, not the member. The 'packed' on
// the outer structure reduces the alignment to 1.
struct C {
  struct {
    char x;
    int y;
  } __attribute__((aligned(2))) z;
} __attribute__((packed));

// the inner structure is anonymous, so the 'aligned' always applies to the
// type, not the member. Structure D is identical to structure C.
struct D {
  struct {
    char x;
    int y;
  } __attribute__((aligned(2)));
} __attribute__((packed));

void test_struct_2() {
  printf("A is size %zd, align %zd\n", sizeof(struct A), _Alignof(struct A));
  printf("  z is at offset %zd\n",  offsetof(struct A, z));
  printf("  z.x is at offset %zd\n",  offsetof(struct A, z) + offsetof(struct S,
 x));
  printf("  z.y is at offset %zd\n",  offsetof(struct A, z) + offsetof(struct S,
 y));

  printf("B is size %zd, align %zd\n", sizeof(struct B), _Alignof(struct B));
  printf("C is size %zd, align %zd\n", sizeof(struct C), _Alignof(struct C));
  printf("D is size %zd, align %zd\n", sizeof(struct D), _Alignof(struct D));

  struct D e __attribute__((aligned(4)));
  printf("e is size %zd, align %zd\n", sizeof(d), _Alignof(e));
}

int main() {
  test_struct_2();
  return 0;
}
```

Compile `test_struct_2.c` using the following command:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -Wno-gnu-alignof-expression
 test_struct_2.c -o test_struct_2.axf -Wall
```

Run the `test_struct_2.axf` image. The output is:

```
A is size 8, align 2
  z is at offset 0
  z.x is at offset 0
  z.y is at offset 4
B is size 8, align 2
C is size 8, align 1
D is size 8, align 1
e is size 4, align 4
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.4 Compiler-specific Intrinsics

Intrinsics that are specific to Arm® Compiler for Embedded, and that are extensions to the C and C ++ Standards.

To use these intrinsics, your source file must contain `#include <arm_compat.h>`.

The available intrinsics are as follows:

- __breakpoint
- __current_pc
- __current_sp
- __disable_fiq
- __disable_irq
- __enable_fiq
- __enable_irq
- __force_stores
- __memory_changed
- __schedule_barrier
- __semihost
- __vfp_status

2.4.1 __breakpoint intrinsic

This intrinsic inserts a `BKPT` instruction into the instruction stream generated by the compiler.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.

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 A32 code

`0 ... 255`
if you are compiling source as T32 code.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Errors

The `__breakpoint` intrinsic is not available when compiling for a target that does not support the `BKPT` instruction. The compiler generates an error in this case.

Example

```
void func(void)
{
    ...
    __breakpoint(0xF02C);
    ...
}
```

2.4.2 __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.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.

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.

2.4.3 __current_sp intrinsic

This intrinsic returns the value of the stack pointer at the current point in your program.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.

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.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.4.4 __disable_fiq intrinsic

This intrinsic disables FIQ interrupts.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.


Note

Typically, this intrinsic disables FIQ interrupts by setting the F-bit in the CPSR. However, for v7-M and v8-M.mainline, it sets the fault mask register (FAULTMASK). This intrinsic is not supported for v6-M and v8-M.baseline.

Syntax

```
int __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.

Return value

`int __disable_fiq(void);` returns the value the FIQ interrupt mask has in the `PSR` before disabling FIQ interrupts.

Restrictions

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();
}
```

2.4.5 __disable_irq intrinsic

This intrinsic disables IRQ interrupts.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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)
```

Usage

`int __disable_irq(void);` disables interrupts and returns the value the IRQ interrupt mask has in the PSR before disabling 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

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.

2.4.6 __enable_fiq intrinsic

This intrinsic enables FIQ interrupts.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.

Note

Typically, this intrinsic enables FIQ interrupts by clearing the F-bit in the CPSR. However, for v7-M and v8-M.mainline, it clears the fault mask register (FAULTMASK). This intrinsic is not supported in v6-M and v8-M.baseline.

Syntax

```
void __enable_fiq(void)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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`.

2.4.7 __enable_irq intrinsic

This intrinsic enables IRQ interrupts.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.



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`.

2.4.8 __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.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.

This intrinsic also acts as a `__schedule_barrier` intrinsic.

Syntax

```
void __force_stores(void)
```

2.4.9 __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.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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.

This intrinsic also acts as a `__schedule_barrier` intrinsic.

Syntax

```
void __memory_changed(void)
```

2.4.10 __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.

Operations with side effects cannot be reordered above or below the `__schedule_barrier` intrinsic. To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.

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)
```

2.4.11 __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.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.

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.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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:


Note

The `HLT` instruction is architecturally `UNDEFINED` for Arm®v7-A and Armv7-R architectures, in both A32 and T32 state.

`SVC 0x123456`
In A32 state, excluding M-profile architectures.

`SVC 0xAB`
In T32 state, excluding M-profile architectures. This behavior is not guaranteed on all debug targets from Arm or from third parties.

`HLT 0xF000`
In A32 state, excluding M-profile architectures.

`HLT 0x3C`
In T32 state, excluding M-profile architectures.

`BKPT 0xAB`
For M-profile architectures (T32 only).

Implementation

For Arm processors that are not Cortex®-M profile, semihosting is implemented using the `SVC` or `HLT` instruction. For Cortex-M profile processors, semihosting is implemented using the `BKPT` instruction.

To use HLT-based semihosting, you must define the pre-processor macro `__USE_HLT_SEMIHOSTING` before `#include <arm_compat.h>`. By default, Arm Compiler for Embedded emits `SVC` instructions rather than `HLT` instructions for semihosting calls. If you define this macro, `__USE_HLT_SEMIHOSTING`, then Arm Compiler for Embedded emits `HLT` instructions rather than `SVC` instructions for semihosting calls.

The presence of this macro, `__USE_HLT_SEMIHOSTING`, does not affect the M-profile architectures that still use `BKPT` for semihosting.

Example

```
char buffer[100];
...
void foo(void)
{
    __semihost(0x01, (const void *)buffer);
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
}
```

Compiling this code with the option `-mthumb` shows the generated SVC instruction:

```
foo:
    ...
    MOVW     r0, :lower16:buffer
    MOVT     r0, :upper16:buffer
    ...
    SVC      #0xab
    ...

buffer:
    .zero    100
    .size    buffer, 100
```

Related information

Using the C and C++ libraries with an application in a semihosting environment

2.4.12 __vfp_status intrinsic

This intrinsic reads or modifies the FPSCR.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for targets in AArch32 state.

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 2-41: 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

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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.

2.5 Compiler-specific Pragmas

Pragmas that are specific to Arm® Compiler for Embedded, and that are extensions to the C and C ++ Standards.

The following pragmas are available:

- #pragma clang diagnostic
- #pragma clang section
- #pragma clang system_header
- #pragma message
- #pragma once
- #pragma pack(...)
- #pragma STDC option value
- #pragma unroll[(n)], #pragma unroll_completely
- #pragma weak symbol, #pragma weak symbol1 = symbol2

2.5.1 #pragma clang diagnostic

This pragma allows you to 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.

Caution

Reducing the severity of diagnostic messages might prevent the tool from reporting important faults. Arm recommends that you do not reduce the severity of diagnostics unless you understand the impact on your software.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

Alternatively, you can use the command-line option, `-W<name>`, to suppress or change the severity of messages, but the change applies for the entire compilation.

Syntax

Disable or set the severity of the specified message:

```
#pragma clang diagnostic <message_type> "-W<name>"
```

Save the current pragma diagnostic state so that it can restored later:

```
#pragma clang diagnostic push
```

Restore the diagnostic state that was previously saved using `#pragma clang diagnostic push`.

```
#pragma clang diagnostic pop
```

Parameters

`<message_type>`

One of:

`ignored`

Disable the specified diagnostic message.

`warning`

Set the specified diagnostic message to warning severity.

`error`

Set the specified diagnostic message to error severity.

`fatal`

Set the specified diagnostic message to fatal error severity.

A fatal error causes compilation to fail without processing the rest of the file.

`<name>`

The diagnostic message to operate on.

Examples: Using pragmas to control diagnostics

The following example shows four identical functions, `foo1()`, `foo2()`, `foo3()`, and `foo4()`. All these functions would normally provoke diagnostic message `warning: multi-character character constant [-Wmultichar]` on the source lines `char c = (char) 'ab';`

Using pragmas, you can suppress or change the severity of these diagnostic messages for individual functions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

For `foo1()`, the current pragma diagnostic state is pushed to the stack and `#pragma clang diagnostic ignored` suppresses the message. The diagnostic message is then re-enabled by `#pragma clang diagnostic pop`.

For `foo2()`, the diagnostic message is not suppressed because the original pragma diagnostic state has been restored.

For `foo3()`, the message is initially suppressed by the preceding `#pragma clang diagnostic ignored "-Wmultichar"`, however, the message is then re-enabled as an error, using `#pragma clang diagnostic error "-Wmultichar"`. The compiler therefore reports an error in `foo3()`.

For `foo4()`, the pragma diagnostic state is restored to the state saved by the preceding `#pragma clang diagnostic push`. This state therefore includes `#pragma clang diagnostic ignored "-Wmultichar"` and therefore the compiler does not report a warning in `foo4()`.

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wmultichar"
void foo1( void )
{
    /* Here we do not expect a diagnostic message, because it is suppressed by
 #pragma clang diagnostic ignored "-Wmultichar". */
    char c = (char) 'ab';
}
#pragma clang diagnostic pop

void foo2( void )
{
    /* Here we expect a warning, because the suppression was inside push and then
 the diagnostic message was restored by pop. */
    char c = (char) 'ab';
}

#pragma clang diagnostic ignored "-Wmultichar"
#pragma clang diagnostic push
void foo3( void )
{
    #pragma clang diagnostic error "-Wmultichar"
    /* Here, the diagnostic message is elevated to error severity. */
    char c = (char) 'ab';
}
#pragma clang diagnostic pop

void foo4( void )
{
    /* Here, there is no diagnostic message because the restored diagnostic state
 only includes the #pragma clang diagnostic ignored "-Wmultichar".
       It does not include the #pragma clang diagnostic error "-Wmultichar" that is
 within the push and pop pragmas. */
    char c = (char) 'ab';
}
```

Diagnostic messages use the pragma state that is present at the time they are generated. If you use pragmas to control a diagnostic message in your code, you must be aware of when, in the compilation process, that diagnostic message is generated.

If a diagnostic message for a function, `functionA`, is only generated after all the functions have been processed, then the compiler controls this diagnostic message using the pragma diagnostic state that is present after processing all the functions. This diagnostic state might be different from the diagnostic state immediately before or within the definition of `functionA`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

-W (armclang) on page 226

2.5.2 #pragma clang section

Specifies names for one or more section types. The compiler places subsequent functions, global variables, or static variables in the named section depending on the section type. The names only apply within the compilation unit.

Syntax

```
#pragma clang section [<section_type_list>]
```

Parameters

<section_type_list>

An optional list of section names to be used for subsequent functions, global variables, or static variables. The syntax of `<section_type_list>` is:

```
<section_type>="<name>"[ <section_type>="<name>"]
```

You can revert to the default section name by specifying an empty string, `""`, for `<name>`.

<section_type>

One of:

- `bss`.
- `data`.
- `relro`.
- `rodata`.
- `text`.

Restrictions

`#pragma clang section` has the following restrictions:

- Each `<section_type>` in the `<section_type_list>` must have a unique name. `armclang` does not permit sections with different type or flags to share the same name.
- Read-only constant initializers, including string literals, are not guaranteed to be placed in the `<section_type> rodata`. The compiler might generate additional sections that contain the initializing data.

Operation

Use `#pragma clang 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.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- If you specify a section name with `__attribute__((section("<myname>")))`, then the attribute name has priority over any applicable section name that you specify with `#pragma clang section`.
- `#pragma clang section` has priority over the `-ffunction-sections` and `-fdata-sections` command-line options.
- Global variables, including basic types, arrays, and struct that are initialized to zero are placed in the `.bss` section. For example, `int x = 0;`.
- `armclang` does not try to infer the type of section from the name. For example, assigning a section `.bss.mysec` does not mean it is placed in a `.bss` section.
- If you specify the `-ffunction-sections` and `-fdata-sections` command-line options, then each global variable is in a unique section. `-ffunction-sections` is the default.
- The position independent code options `-fpic` and `-fpie` do not support relocations in read-only code. The compiler uses a relocation read only (RELRO) section that is read-write but has a special name with a prefix of `.data.rel.ro` to support code such as:

  ```
  int rw;
  int read_write = 10;
  int * const ro_pointer_to_rw = &rw;
  ```

  You can rename these sections using `relro="<name>"`.

The compiler produces an error if:

- You use the same section name for different section types.
- You use a section name that is the same as a variable, function, or other symbol in your program.

Example: Placement of functions and data

```
int x1 = 5;                     // Goes in .data section (default)
int y1;                         // Goes in .bss section (default)
const int z1 = 42;              // Goes in .rodata section (default)
char *s1 = "abc1";              // s1 goes in .data section (default). String "abc1"
 goes in .conststring section.

#pragma clang section bss="myBSS" data="myData" rodata="myRodata"
int x2 = 5;                     // Goes in myData section.
int y2;                         // Goes in myBss section.
const int z2 = 42;              // Goes in myRodata section.
char *s2 = "abc2";              // s2 goes in myData section. String "abc2" goes
 in .conststring section.

#pragma clang section rodata=""   // Use default name for rodata section.
int x3 = 5;                     // Goes in myData section.
int y3;                         // Goes in myBss section.
const int z3 = 42;              // Goes in .rodata section (default).
char *s3 = "abc3";              // s3 goes in myData section. String "abc3" goes
 in .conststring section.

#pragma clang section text="myText"
int add1(int x)                 // Goes in myText section.
{
    return x+1;
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
#pragma clang section bss="" data="" text="" // Use default name for bss, data, and
 text sections.
```

Example: Placement of a relro section

1. Create a C file, `test.c`, containing:

```
int rw;
int read_write = 10;
int * const ro_pointer_to_rw = &rw;
```

2. Compile the file with:

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -fpic -o relro.o test.c
```

3. Run `fromelf` to view the sections in the `.o` file:

```
fromelf -s relro.o

...
** Section #5 '.data.rel.ro.ro_pointer_to_rw' (SHT_PROGBITS) [SHF_ALLOC +
 SHF_WRITE]
    Size   : 4 bytes (alignment 4)
    Address: 0x00000000


** Section #6 '.rel.data.rel.ro.ro_pointer_to_rw' (SHT_REL) [SHF_INFO_LINK]
    Size   : 8 bytes (alignment 4)
    Symbol table #10 '.symtab'
    1 relocations applied to section #5 '.data.rel.ro.ro_pointer_to_rw'
...
```

   You can see that the `.data.rel.ro` section is used.

4. Compile again without the `-fpic` option:

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -o relro.o test.c
```

   The `fromelf` output now shows that `rodata` is used:

```
fromelf -s relro.o

...
** Section #5 'myrodata' (SHT_PROGBITS) [SHF_ALLOC]
    Size   : 4 bytes (alignment 4)
    Address: 0x00000000


** Section #6 '.relmyrodata' (SHT_REL) [SHF_INFO_LINK]
    Size   : 8 bytes (alignment 4)
    Symbol table #10 '.symtab'
    1 relocations applied to section #5 'myrodata'
...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.5.3 #pragma clang system_header

This pragma causes subsequent declarations in the current file to be marked as if they occur in a system header file.

The pragma suppresses the warning messages that the file produces, from the point after which it is declared.

Syntax

```
#pragma clang system_header
```

Parameters

None.

2.5.4 #pragma message

Generates your own custom message during compilation. The message is output as a warning.

Syntax

```
#pragma message <message>
```

Parameters

`<message>`

The message to output as a compiler message. You can output multiple lines with the `'\n'` escape sequence. You can also use predefined macros in the message.

You can parenthesize `<message>` by embedding the C99 `_Pragma(<string-literal>)` operator in a macro. `<string-literal>` can be either a normal or wide-character string literal.

Example: Output a compiler message containing a predefined macro

Create the file `test.c` containing:

```
#pragma message "Compiling " __FILE_NAME__ "..."
```

Compile with:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a8 -c test.c
```

The following warning is output:

```
test.c:1:9: warning: Compiling test.c... [-W#pragma-messages]
    1 | #pragma message "Compiling " __FILE_NAME__ "..."
      |         ^
1 warning generated.
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The first line of the warning message shows the specified message with the predefined macro expanded.

Example: Output a compiler message by embedding the _Pragma operator in a macro

Create the file `test1.c` containing:

```
#define DO_PRAGMA(x) _Pragma(#x)
#define TODO(x) DO_PRAGMA(message ("TODO - " #x))

TODO(Remember to fix this)
```

Compile with:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a8 -c test1.c
```

The following warning and note messages are output:

```
test1.c:4:1: warning: TODO - Remember to fix this [-W#pragma-messages]
    4 | TODO(Remember to fix this)
      | ^
test1.c:2:17: note: expanded from macro 'TODO'
    2 | #define TODO(x) DO_PRAGMA(message ("TODO - " #x))
      |                 ^
test1.c:1:22: note: expanded from macro 'DO_PRAGMA'
    1 | #define DO_PRAGMA(x) _Pragma (#x)
      |                      ^
<scratch space>:4:2: note: expanded from here
    4 |  message ("TODO - " "Remember to fix this")
      |  ^
1 warning generated.
```

The first line of the warning message shows the generated message.

Related information

Predefined macros on page 336

2.5.5 #pragma once

Enable 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, Arm recommends using `#ifndef` and `#define` coding because this is more portable.

Syntax

```
#pragma once
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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.

2.5.6 #pragma pack(...)

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. You can optionally push and restore alignment settings to an internal stack.


Note

This pragma is a GNU compiler extension that the Arm® Compiler for Embedded supports.

Default

The default is the alignment that was in effect when compilation started.

Syntax

```
#pragma pack([<n>])
#pragma pack(push[,<n>])
#pragma pack(pop)
```

Parameters

<n>

Is the alignment in bytes, valid alignment values are `1`, `2`, `4`, and `8`. If omitted, sets the alignment to the one that is in effect at the start of the compilation.

push[,<n>]

Pushes the current alignment setting on an internal stack and then optionally sets the new alignment.

pop

Restores the alignment setting to the one saved at the top of the internal stack, then removes that stack entry.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

`#pragma pack([<n>])` does not influence this internal stack. Therefore, it is possible to have `#pragma pack(push)` followed by multiple `#pragma pack([<n>])` instances, then finalized by a single `#pragma pack(pop)`.

Example

This example shows how `pack(2)` aligns integer variable `b` to a 2-byte boundary.

```
typedef struct
{
    char a;
    int b;
} S;

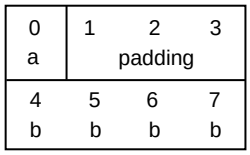
#pragma pack(2)

typedef struct
{
    char a;
    int b;
} SP;

S var = { 0x11, 0x44444444 };
SP pvar = { 0x11, 0x44444444 };
```

The layout of `s` is:

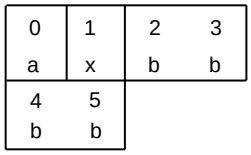
Figure 2-5: Nonpacked structure S



The layout of `SP` is:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 2-6: Packed structure SP



In this layout, `x` denotes one byte of padding.

`SP` is a 6-byte structure. There is no padding after `b`.

2.5.7 #pragma STDC

C standard pragmas for *floating-point* (FP).

Default

The default value of the `FENV_ACCESS` pragma is `OFF`.

The default value of the `FP_CONTRACT` pragma depends on the value set by the `-ffp-contract=<value>` command-line option.

`-ffp-mode`	`-ffp-contract` default	`#pragma STDC FP_CONTRACT`
`fast`	`fast`	`#pragma STDC FP_CONTRACT OFF` has no effect because `#pragma STDC FP_CONTRACT` is always `ON`.
`std` or `full`	`on`	`#pragma STDC FP_CONTRACT OFF` overrides `-ffp-contract=on`.

Syntax

```
#pragma STDC <option> <value>
```

Parameters

`<option>`

`FENV_ACCESS`

Indicates the FP environment access.

`FP_CONTRACT`

Indicates whether the compiler can perform FP expression contractions.

`<value>`

Either `ON` or `OFF`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Restrictions

The value `ON` is not supported for `FENV_ACCESS`.

Example: FP_CONTRACT examples

The `FP_CONTRACT` examples show how you can use `-ffp-contract=<value>` and `#pragma STDC FP_CONTRACT <value>` together when compiling with `-ffp-mode=std` to change the behavior of the compiler in different ways.

The examples are available in #pragma STDC FP_CONTRACT examples.

Related information

-ffp-contract on page 61
Floating-point on page 1051

2.5.8 #pragma unroll[(n)], #pragma unroll_completely

Instructs the compiler to unroll a loop by `<n>` iterations.

Default

If you do not specify a value for `<n>`, the compiler attempts to fully unroll the loop. The compiler can only fully unroll loops where it can determine the number of iterations.

`#pragma unroll_completely` does not unroll a loop if the number of iterations is not known at compile time.

Syntax

```
#pragma unroll
#pragma unroll_completely
#pragma unroll <n>
#pragma unroll(<n>)
```

Parameters

<n>

An optional value indicating the number of iterations to unroll.

Restrictions

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.

Operation

This pragma is supported at the `-Os`, `-Oz`, `-O2`, `-O3`, `-Ofast`, and `-Omax` optimization levels. It is not supported at the `-O0` or `-O1` optimization levels.

When compiling at `-O3`, the compiler automatically unrolls loops where it is beneficial to do so. This pragma can be used to ask the compiler to unroll a loop that has not been unrolled automatically.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`#pragma unroll[(<n>)]` can be used immediately before a for loop, a while loop, or a do ... while loop.

2.5.9 #pragma weak symbol, #pragma weak symbol1 = symbol2

This pragma is a language extension to mark symbols as weak or to define weak aliases of symbols.

Syntax

```
#pragma weak <symbol>
#pragma weak <symbol> = <symbol2>
```

Parameters

`<symbol>`

A symbol that you want to define as weak, or the name that you want to use as an alias for a weak symbol.

`<symbol2>`

The symbol for which you want to define a weak alias.

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)
{
    ...
}
```

2.5.10 #pragma STDC FP_CONTRACT examples

The `FP_CONTRACT` examples show how you can use `-ffp-contract=<value>` and `#pragma STDC FP_CONTRACT <value>` together when compiling with `-ffp-mode=std` to change the behavior of the compiler in different ways.

For more information about the `#pragma STDC FP_CONTRACT` pragma, see #pragma STDC.

About the FP_CONTRACT examples

The following examples use the file `functions.c` that contains the following C code:

```
// functions.c

float fn1(float a, float b, float c) {
    // fmadd with -ffp-contract=on
    // fmul, fadd with -ffp-contract=off
    return a*b + c;
}
float fn2(float a, float b, float c) {
    #pragma STDC FP_CONTRACT ON
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    // fmadd
    return a*b + c;
}
float fn3(float a, float b, float c) {
    #pragma STDC FP_CONTRACT OFF
    // fmul, fadd unless -ffp-contract=fast
    return a*b + c;
}
```

For more examples showing the interaction of the `FP_CONTRACT` pragma and the `-ffp-contract=<value>` option, see the examples for -ffp-contract.

Example: -ffp-contract=on

Compile `functions.c` with the following command:

```
armclang -c --target=aarch64-arm-none-eabi -march=armv8-a -O1 -ffp-mode=std -S -o
 ffp-contract_on.s functions.c
```

Because `-ffp-mode=std` is specified, `-ffp-contract=on` is the default. Therefore, the following assembly code is generated for each function:

```
...
fn1:
    fmadd  s0, s0, s1, s2
    ret
...
fn2:
    fmadd  s0, s0, s1, s2
    ret
...
fn3:
    fmul  s0, s0, s1
    fadd  s0, s0, s2
    ret
...
```

The result of this example is as follows:

Function	Operation fused (Y/N)
`fn1()`	Y
`fn2()`	Y
`fn3()`	N

Notice that `#pragma STDC FP_CONTRACT OFF` overrides `-ffp-contract=on`.

Example: -ffp-contract=off

Compile `functions.c` with `-ffp-contract=off`:

```
armclang -c --target=aarch64-arm-none-eabi -march=armv8-a -O1 -ffp-mode=std -ffp-
contract=off -S -o ffp-contract_on.s functions.c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The following assembly is generated for each function:

```
...
fn1:
    fmul  s0, s0, s1
    fadd  s0, s0, s2
    ret
...
fn2:
    fmadd  s0, s0, s1, s2
    ret
...
fn3:
    fmul  s0, s0, s1
    fadd  s0, s0, s2
    ret
...
```

The result of this example is as follows:

Function	Operation fused (Y/N)
`fn1()`	N
`fn2()`	Y
`fn3()`	N

Notice that `#pragma STDC FP_CONTRACT ON` overrides `-ffp-contract=off`.

Example: -ffp-contract=fast

Compile `functions.c` with `-ffp-contract=fast`:

```
armclang -c --target=aarch64-arm-none-eabi -march=armv8-a -O1 -ffp-mode=std -ffp-
contract=fast -S -o ffp-contract_on.s functions.c
```

The following assembly is generated for each function:

```
...
fn1:
    fmadd  s0, s0, s1, s2
    ret
...
fn2:
    fmadd  s0, s0, s1, s2
    ret
...
fn3:
    fmadd  s0, s0, s1, s2
    ret
...
```

The result of this example is as follows:

Function	Operation fused (Y/N)
`fn1()`	Y
`fn2()`	Y

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Function	Operation fused (Y/N)
`fn3()`	Y

Notice that `#pragma STDC FP_CONTRACT OFF` has no effect, because for `-ffp-contract=fast` the `FP_CONTRACT` pragma is always `ON`.

Related information

-ffp-contract on page 61
-ffp-mode on page 62

2.6 Other Compiler-specific Features

Features that are specific to Arm® Compiler for Embedded, and that are extensions to the C and C ++ Standards, such as predefined macros.

2.6.1 ACLE support

The intrinsics you can use in Arm® Compiler for Embedded 6 are described in the *Arm C Language Extensions* (ACLE) specification and supplementary documents.



Note

This topic includes descriptions of [ALPHA] and [BETA] features. See Support level definitions.

You can download the latest version of the ACLE from GitHub.

The ACLE standardizes intrinsics to access the Arm extensions. The ACLE header files list the supported intrinsics. These files are located in the `<install_directory>/include` directory, for example:

- `arm_acle.h` defines the non-Neon intrinsics.
- `arm_neon.h` defines the Neon intrinsics.
- `arm_mve.h` defines the *M-profile Vector Extension* (MVE) intrinsics.
- `arm_cde.h` defines the *Custom Datapath Extension* (CDE) intrinsics.
- `arm_sve.h` defines the SVE and SVE2 intrinsics.
- [ALPHA] `arm sme.h` defines the SME and SME2 intrinsics.



Note

The `arm_sve.h` header file is not supported for big-endian targets.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Arm Compiler for Embedded 6 does not support:

- `__ARM_ALIGN_MAX_PWR` macro.
- `__ARM_ALIGN_MAX_STACK_PWR` macro.
- `__saturation_occurred` intrinsic.
- `__set_saturation_occurred` intrinsic.
- `__ignore_saturation` intrinsic.
- Patchable constants.
- Floating-point data-processing intrinsics.

Arm Compiler for Embedded 6 does not model the state of the Q (saturation) flag correctly in all situations.

Additional supported intrinsics

Arm Compiler for Embedded 6 also provides:

- Support for the ACLE defined dot product intrinsics in AArch64 and AArch32 states.
- [BETA] Support for the ACLE defined Armv8.2-A half-precision floating-point scalar and vector intrinsics in AArch64 state.
- [BETA] Support for the ACLE defined Armv8.2-A half-precision floating-point vector intrinsics in AArch32 state.
- Support for the ACLE defined BFloat16 floating-point scalar and vector intrinsics in AArch64 and AArch32 states.
- Support for the ACLE defined Matrix Multiplication scalar and vector intrinsics in AArch64 and AArch32 states.
- Support for the ACLE defined *Memory Tagging Extension* (MTE) intrinsics.
- Support for the ACLE defined *Transactional Memory Extension* (TME) intrinsics.
- Support for the ACLE defined *M-profile Vector Extension* (MVE) intrinsics. For more information on the MVE intrinsics, see C language extensions and the Helium Intrinsics.
- Support for the ACLE defined Special register intrinsics:
  - `__arm_rsrf`
  - `__arm_wsrf`
  - `__arm_rsrf64`
  - `__arm_wsrf64`
  - `__arm_rsr128`
  - `__arm_wsr128`

Related information

Half-precision floating-point intrinsics on page 347
Neon intrinsics
Helium intrinsics

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Differences between various standard modes

2.6.2 Predefined macros

Arm® Compiler for Embedded predefines a number of macros. These macros provide information about toolchain version numbers and compiler options.

In general, the predefined macros generated by the compiler are compatible with those generated by GCC. See the GCC documentation for more information.

The following table lists Arm-specific macro names predefined by Arm Compiler for Embedded for C and C++, together with a number of the most commonly used macro names. Where the value field is empty, the symbol is only defined.


Note

Use `-E -dM` to see the values of predefined macros.

Macros beginning with `__ARM_` are defined by the *Arm C Language Extensions 2.1* (ACLE 2.1). For more information see Arm C Language Extensions (ACLE).


Note

`armclang` does not fully implement ACLE 2.1.

Table 2-46: Predefined macros

Name	Value	When defined
`__APCS_ROPI`	`1`	Set when you specify the `-fropi` option.
`__APCS_RWPI`	`1`	Set when you specify the `-frwpi` option.
`__ARM_64BIT_STATE`	`1`	Set for targets in AArch64 state only. Set to `1` if code is for 64-bit state.
`__ARM_ALIGN_MAX_STACK_PWR`	`4`	Set for targets in AArch64 state only. The log of the maximum alignment of the stack object.
`__ARM_ARCH`	`<ver>`	Specifies the version of the target architecture. For example, `8` means Armv8.
`__ARM_ARCH_EXT_IDIV__`	`1`	Set for targets in AArch32 state only. Set to `1` if hardware divide instructions are available.
`__ARM_ARCH_ISA_A64`	`1`	Set for targets in AArch64 state only. Set to `1` if the target supports the A64 instruction set.
`__ARM_ARCH_PROFILE`	`<ver>`	Specifies the profile of the target architecture, for example 'A'.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Name	Value	When defined
__ARM_BIG_ENDIAN	-	Set if compiling for a big-endian target.
__ARM_FEATURE_BTI_DEFAULT	1	Defined if the target enforcement is enabled with the `-mbranch-protection=bti` option.
__ARM_FEATURE_PAC_DEFAULT	Depends on the options specified for `-mbranch-protection=<protection>`.	Defined if the return address signing and authentication is enabled with the `-mbranch-protection=pac-ret` option.
__ARM_FEATURE_CLZ	1	Set to 1 if the count leading zeroes, `CLZ`, instruction is supported in hardware.
__ARM_FEATURE_CMSE	<num>	Indicates the availability of the Armv8-M Security Extension related instructions: **0** The `TT` and `TTA` instructions are not available. **1** The `TT` instruction is supported. **3** The `TT` and `TTA` instructions are supported. See -mcmse and TT instruction intrinsics for more information.
__ARM_FEATURE_CRC32	1	Set to 1 if the target has `CRC` extension.
__ARM_FEATURE_CRYPTO	1	Set to 1 if the target has cryptographic extension.
__ARM_FEATURE_DIRECTED_ROUNDING	1	Set to 1 if the directed rounding and conversion vector instructions are supported. Only available when `__ARM_ARCH >= 8`.
__ARM_FEATURE_DSP	1	Set for targets in AArch32 state only. Set to 1 if DSP instructions are supported. This feature also implies support for the Q flag. **Note:** This macro is deprecated for A-profile. It is fully supported for M and R-profiles.
__ARM_FEATURE_IDIV	1	Set to 1 if the target supports 32-bit signed and unsigned integer division in all available instruction sets.
__ARM_FEATURE_FMA	1	Set to 1 if the target supports fused floating-point multiply-accumulate.
__ARM_FEATURE_MOPS	1	Set to 1 if the target supports A-profile Memory Operations Extension.
__ARM_FEATURE_NUMERIC_MAXMIN	1	Set to 1 if the target supports floating-point maximum and minimum instructions. Only available when `__ARM_ARCH >= 8`.
__ARM_FEATURE_PAUTH	1	Set to 1 if the target supports Pointer Authentication extension.
__ARM_FEATURE_PAUTH_LR	1	Set to 1 if the target supports A-profile Enhancements to Pointer Authentication extension.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Name	Value	When defined
`__ARM_FEATURE_QBIT`	1	Set for targets in AArch32 state only. Set to 1 if the saturation flag Q exists. **Note:** This macro is deprecated for A-profile.
`__ARM_FEATURE_SAT`	1	Set for targets in AArch32 state only. Set to 1 if the `SSAT` and `USAT` instructions are supported. This feature also implies support for the Q flag. **Note:** This macro is deprecated for A-profile.
`__ARM_FEATURE_SIMD32`	1	Set for targets in AArch32 state only. Set to 1 if the target supports 32-bit SIMD instructions. **Note:** This macro is deprecated for A-profile, use Arm® Neon® intrinsics instead.
`__ARM_FEATURE_SYSREG128`	1	Set for targets in AArch64 state only. Set to 1 if the `__arm_rsr128` and `__arm_wsr128` intrinsics are available.
`__ARM_FEATURE_UNALIGNED`	1	Set to 1 if the target supports unaligned access in hardware.
`__ARM_FP`	`<val>`	Set if hardware floating-point is available. Bits 1-3 indicate the supported floating-point precision levels. The other bits are reserved. • Bit 1 - half precision (16-bit). • Bit 2 - single precision (32-bit). • Bit 3 - double precision (64-bit). These bits can be bitwise or-ed together. Permitted values include: • `0x04` for single-support. • `0x0C` for single- and double-support. • `0x0E` for half-, single-, and double-support.
`__ARM_FP_FAST`	1	Set if `-ffast-math` or `-ffp-mode=fast` is specified.
`__ARM_NEON`	1	Set to 1 when the compiler is targeting an architecture or processor with Advanced SIMD available. Use this macro to conditionally include `arm_neon.h`, to permit the use of Advanced SIMD intrinsics.
`__ARM_NEON_FP`	`<val>`	This is the same as `__ARM_FP`, except that the bit to indicate double-precision is not set for targets in AArch32 state. Double-precision is always set for targets in AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Name	Value	When defined
__ARM_PCS	1	Set for targets in AArch32 state only. Set to 1 if the default procedure calling standard for the translation unit conforms to the base PCS.
__ARM_PCS_VFP	1	Set for targets in AArch32 state only. Set to 1 if the default procedure calling standard for the translation unit conforms to the VFP PCS. That is, `-mfloat-abi=hard`.
__ARM_SIZEOF_MINIMAL_ENUM	<value>	Specifies the size of the minimal enumeration type. Set to either 1 or 4 depending on whether `-fshort-enums` is specified or not.
__ARM_SIZEOF_WCHAR_T	value	Specifies the size of `wchar` in bytes. Set to: • 2 if `-fshort-wchar` is specified • 4 if `-fno-short-wchar` is specified. **Note:** The default size is 4, because `-fno-short-wchar` is set by default.
__ARMCOMPILER_VERSION	<Mmmuuxx>	Always set. Specifies the version number of the compiler, `armclang`. The format is <Mmmuuxx>, where: <M> is the major version number, 6. <mm> is the minor version number. <uu> is the update number. <xx> is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions. For example, version 6.16.1 is displayed as 6160154, where 54 is a number for Arm internal use.
__ARMCC_VERSION	<Mmmuuxx>	A synonym for __ARMCOMPILER_VERSION.
__arm__	1	Defined when targeting AArch32 state with: `--target=arm-arm-none-eabi` See also __aarch64__.
__aarch64__	1	Defined when targeting AArch64 state with: `--target=aarch64-arm-none-eabi` See also __arm__.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Name	Value	When defined
`__cplusplus`	`<ver>`	Defined when compiling C++ code, and set to a value that identifies the targeted C++ standard. For example, when compiling with `-xc++ -std=gnu++98`, the compiler sets this macro to `199711L`. You can use the `__cplusplus` macro to test whether a file was compiled by a C compiler or a C++ compiler.
`__CHAR_UNSIGNED__`	1	Defined if and only if `char` is an unsigned type.
`__EXCEPTIONS`	1	Defined when compiling a C++ source file with exceptions enabled.
`__FILE_NAME__`	`name`	Contains the filename part of the value of `__FILE__`.
`__GNUC__`	`<ver>`	Always set. An integer that specifies the major version of the compatible GCC version. This macro indicates that the compiler accepts GCC compatible code. The macro does not indicate whether the `-std` option has enabled GNU C extensions. For detailed Arm Compiler for Embedded version information, use the `__ARMCOMPILER_VERSION` macro.
`__INTMAX_TYPE__`	`<type>`	Always set. Defines the correct underlying type for the `intmax_t typedef`.
`__NO_INLINE__`	1	Defined if no functions have been inlined. The macro is always defined with optimization level `-O0` or if the `-fno-inline` option is specified.
`__OPTIMIZE__`	1	Defined when `-O1`, `-O2`, `-O3`, `-Ofast`, `-Oz`, or `-Os` is specified.
`__OPTIMIZE_SIZE__`	1	Defined when `-Os` or `-Oz` is specified.
`__PTRDIFF_TYPE__`	`<type>`	Always set. Defines the correct underlying type for the `ptrdiff_t typedef`.
`__SIZE_TYPE__`	`<type>`	Always set. Defines the correct underlying type for the `size_t typedef`.
`__SOFTFP__`	1	Defined depending on the `-mfloat-abi` value and whether a target has hardware floating-point support. See When the __SOFTFP__ predefined macro is defined.
`__STDC__`	1	Always set. Signifies that the compiler conforms to ISO Standard C.
`__STRICT_ANSI__`	1	Defined if you specify the `--ansi` option or specify one of the `--std=c*` options.
`__thumb__`	1	Defined if you specify the `-mthumb` option.
`__UINTMAX_TYPE__`	`<type>`	Always set. Defines the correct underlying type for the `uintmax_t typedef`.
`__VERSION__`	`<ver>`	Always set. A string that shows the underlying Clang version.
`__WCHAR_TYPE__`	`<type>`	Always set. Defines the correct underlying type for the `wchar_t typedef`.
`__WINT_TYPE__`	`<type>`	Always set. Defines the correct underlying type for the `wint_t typedef`.

When the __SOFTFP__ predefined macro is defined

`__SOFTFP__` is defined as follows:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 2-47: __SOFTFP__ predefined macro

`-mfloat-abi=<value>`	Targets with hardware floating-point support	Targets without hardware floating-point support
Default	`__SOFTFP__` not defined	`__SOFTFP__` defined and set to 1
hard	`__SOFTFP__` not defined	`__SOFTFP__` not defined
soft	`__SOFTFP__` defined and set to 1	`__SOFTFP__` defined and set to 1
softfp	`__SOFTFP__` not defined	`__SOFTFP__` defined and set to 1

Related information

armclang Command-line Options on page 35

2.6.3 Inline functions

Inline functions offer a trade-off between code size and performance. By default, the compiler decides whether to inline functions.

With regards to optimization, by default the compiler optimizes for performance with respect to time. If the compiler decides to inline a function, it makes sure to avoid large code growth. When compiling to restrict code size, by using `-Oz` or `-Os`, the compiler makes sensible decisions about inlining and aims to keep code size to a minimum.

In most circumstances, the decision to inline a particular function is best left to the compiler. Assigning the `__inline__` or `inline` keyword to a function suggests to the compiler that it inlines that function, but the final decision rests with the compiler. Assigning `__attribute__((always_inline))` to a function forces the compiler to inline that function.

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


Note

The default semantic rules for C-source code follow C99 rules. When suggesting a function is inlined, then for inlining, the compiler expects to find an equivalent implementation of the function that does not use `inline`. The compiler uses this equivalent implementation when it decides not to inline. If the compiler cannot find the equivalent implementation, it fails with the following error:

```
"Error: L6218E: Undefined symbol <symbol> (referred from <file>)"
```

To avoid this problem, there are several options:

- Provide an equivalent implementation of the function.
- Change the `inline` or `__inline__` keyword to `static inline`.
- Remove the `inline` or `__inline__` keyword, because it is only acting as a suggestion.
- Compile your program using the GNU C90 dialect, using the `-std=gnu90` option.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

__inline on page 250
-std on page 218
__attribute__((always_inline)) function attribute on page 259

2.6.4 Volatile variables

Arm® Compiler for Embedded does not guarantee that a single-copy atomic instruction is used to access a `volatile` variable that is larger than the natural architecture data size, even when one is available for the target processor.

When compiling for AArch64 state, the natural architecture data size is 64-bits. Targets such as the Cortex®-A53 processor support single-copy atomic instructions for 128-bit data types. In this case, you might expect the compiler to generate an instruction with single-copy atomicity to access a `volatile` 128-bit variable. However, the architecture does not guarantee single-copy atomicity access. Therefore, the compiler does not support it.

When compiling for AArch32 state, the natural architecture data size is 32-bits. In this case, you might expect the compiler to generate an instruction with single-copy atomicity to access a `volatile` 64-bit variable. However, the architecture does not guarantee single-copy atomicity access. Therefore, the compiler does not support it.

Related information

Effect of the volatile keyword on compiler optimization
Arm Architecture Reference Manual for A-profile architecture
ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition

2.6.5 Half-precision floating-point data types

Use the `_Float16` data type for 16-bit floating-point values in your C and C++ source files.

Arm® Compiler for Embedded 6 supports two half-precision (16-bit) floating-point scalar data types:

- The IEEE 754-2008-compliant `__fp16` data type, defined in the *Arm C Language Extensions* (ACLE).
- The `_Float16` data type, defined in the C11 extension ISO/IEC TS 18661-3:2015



Note

Using `_Float16` with `-ffp-mode=full` is not supported. `armclang` does not emit a diagnostic when using `_Float16` with `-ffp-mode=full`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

Arm Compiler for Embedded 6 does not support the flushing of denormal half-precision values.

The `__fp16` data type is not an arithmetic data type. The `__fp16` data type is for storage and conversion only. Operations on `__fp16` values do not use half-precision arithmetic. The values of `__fp16` automatically promote to single-precision `float` (or double-precision `double`) floating-point data type when used in arithmetic operations. After the arithmetic operation, these values are automatically converted to the half-precision `__fp16` data type for storage. The `__fp16` data type is available in both C and C++ source language modes.

The `_Float16` data type is an arithmetic data type. Operations on `_Float16` values use half-precision arithmetic. The `_Float16` data type is available in both C and C++ source language modes.

We recommend that for new code, you use the `_Float16` data type instead of the `__fp16` data type. `__fp16` is an Arm C language extension and therefore requires compliance with the ACLE. `_Float16` is defined by the C standards committee, and therefore using `_Float16` does not prevent code from being ported to architectures other than Arm. Also, `_Float16` arithmetic operations directly map to Armv8.2-A half-precision floating-point instructions when they are enabled on Armv8.2-A and later architectures. This avoids the need for conversions to and from single-precision floating-point, and therefore results in more performant code. If the Armv8.2-A half-precision floating-point instructions are not available, `_Float16` values are automatically promoted to single-precision, similar to the semantics of `__fp16` except that the results continue to be stored in single-precision floating-point format instead of being converted back to half-precision floating-point format.

To define a `_Float16` literal, append the suffix `f16` to the compile-time constant declaration. There is no default argument promotion between `_Float16` and standard floating-point data types. Therefore, an explicit cast is required for promoting `_Float16` to a single-precision floating-point format, for argument passing when default argument promotion is expected.

```
extern void ReadFloatValue(float f);
extern void ReadFloatEllipsis(float f, ...);
void ReadValues(void)
{
    // Half-precision floating-point values stored in the _Float16 data type.
    const _Float16 h1 = 1.0f16;
    const _Float16 h2 = 1.0f16;

    // ReadFloatValue has a prototype, h1 is implicitly converted to float.
    ReadFloatValue(h1);

    // No function prototype. No argument promotion, h1 remains _Float16.
    ReadFloatNoPrototype(h1);

    // h1 is impicitly converted to float, no argument promotion for h2, it remains
 _Float16.
    ReadFloatEllipsis(h1, h2);

    // If the function with no prototype or an ellipsis expects argument promotion,
 an explicit cast is required.
    ReadFloatEllipsis(h1, (double)h2);

    return;
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
}
```

In an arithmetic operation where one operand is of `__fp16` data type and the other is of `_Float16` data type, the `_Float16` value is first converted to `__fp16` value and then the operation is completed as if both operands were of `__fp16` data type.

```
void AddValues(_Float16 a, __fp16 b)
{
    _Float16 c;
    __fp16 d;

    // This addition is evaluated in 16-bit half-precision arithmetic.
    // The result is stored in 16 bits using the _Float16 data type.
    c = a+a;

    // This addition is evaluated in 32-bit single-precision arithmetic.
    // The result is stored in 16 bits using the __fp16 data type.
    d = b+b;

    // The value in variable 'a' in this addition is converted to a __fp16 value.
    // And then the addition is evaluated in 32-bit single-precision arithmetic.
    // The result is stored in 16 bits using the __fp16 data type.
    d = a+b;

    return;
}
```

To generate Armv8.2 half-precision floating-point instructions using `armclang`, you must use the `+fp16` architecture extension, for example:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.2-a+fp16
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a75+fp16
armclang --target=arm-arm-none-eabi -march=armv8.2-a+fp16
armclang --target=arm-arm-none-eabi -mcpu=cortex-a75+fp16
```

Example: Differences between __fp16 and _Float16 at -O1 and higher

Create the file `fp16.c` containing the following code:

```
_Float16 add_float16(_Float16 a, _Float16 b, _Float16 c) { return a + b + c; }
__fp16   add_fp16   (__fp16 a,   __fp16 b,   __fp16 c)   { return a + b + c; }
```

Compile the code with the command:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.2-a+fp16 -c fp16.c -o - -S -O1
```

The command generates the following assembly code, with the parts removed that are not essential for the example:

```
...
add_float16:                            // @add_float16
        fadd    h0, h0, h1
        fadd    h0, h0, h2
        ret
.Lfunc_end0:
...
add_fp16:                               // @add_fp16
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
        fcvt    s0, h0
        fcvt    s1, h1
        fadd    s0, s0, s1
        fcvt    s1, h2
        fadd    s0, s0, s1
        fcvt    h0, s0
        ret
...
```

With `__fp16`, the intermediate value in the expression is of type `float`, so must not be rounded to half-precision. With `_Float16`, the intermediate value in `h0` is of type `_Float16`, so it must be rounded to half-precision. This rounding has the effect that, if the FP16 hardware extension is not enabled, the `_Float16` version must insert extra conversions to ensure that rounding does happen.

To see the difference, compile the code again, but without `+fp16`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.2-a -c fp16.c -o - -S -O1
```

The command generates the following assembly code:

```
...
add_float16:                            // @add_float16
        fcvt    s1, h1
        fcvt    s0, h0
        fadd    s0, s0, s1
        fcvt    s1, h2
        fcvt    h0, s0
        fcvt    s0, h0
        fadd    s0, s0, s1
        fcvt    h0, s0
        ret
.Lfunc_end0:
...
add_fp16:                               // @add_fp16
        fcvt    s0, h0
        fcvt    s1, h1
        fadd    s0, s0, s1
        fcvt    s1, h2
        fadd    s0, s0, s1
        fcvt    h0, s0
        ret
...
```

Related information

-march on page 123
-mcpu on page 150
-ffp-mode on page 62
Library support for _Float16 data type on page 348
Using Assembly and Intrinsics in C or C++ Code
List of intrinsics
Arm C Language Extensions ACLE Q1 2019

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.6.6 Half-precision floating-point number format

Arm® Compiler for Embedded supports the half-precision floating-point `__fp16` type.

Half-precision is a floating-point format that occupies 16 bits. Architectures that support half-precision floating-point values include:

- The Armv8 architecture.
- The Armv7 FPv5 architecture.
- The Armv7 VFPv4 architecture.
- The Armv7 VFPv3 architecture (as an optional extension).

Note

Arm Compiler for Embedded 6 does not support the flushing of denormal half-precision values.

If the target hardware does not support half-precision floating-point values, the compiler uses the floating-point library `fplib` to provide software support for half-precision.

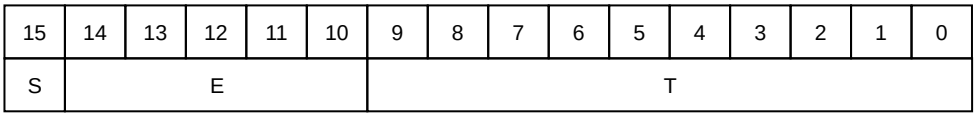
Note

The `__fp16` type is a storage format only. For purposes of arithmetic and other operations, `__fp16` values in C or C++ expressions are automatically promoted to `float`.

Half-precision floating-point format

Arm Compiler for Embedded uses the half-precision binary floating-point format defined by IEEE 754r, a revision to the IEEE 754 standard:

Figure 2-7: IEEE half-precision floating-point format



Where:

```
S (bit[15]):      Sign bit
E (bits[14:10]):  Biased exponent
T (bits[9:0]):    Mantissa.
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The meanings of these fields are 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))
```

Note

See the *Arm C Language Extensions* for more information.

Related information

Arm C Language Extensions ACLE Q1 2019

2.6.7 Half-precision floating-point intrinsics

Arm® Compiler for Embedded 6 provides [BETA] support for the ACLE defined Armv8.2-A half-precision floating-point scalar and vector intrinsics in AArch64 state, and half-precision floating-point vector intrinsics in AArch32 state.

Note

This topic describes a [BETA] feature. See Support level definitions.

To see the half-precision floating-point intrinsics, you can search for `float16` from the list of intrinsics on Neon Intrinsics.

`arm_neon.h` defines the intrinsics for the vector half-precision floating-point intrinsics.

`arm_fp16.h` defines the intrinsics for the scalar half-precision floating-point intrinsics.

The example below demonstrates the use of the half-precision floating-point intrinsics in AArch64 state.

```
// foo.c
#include <arm_neon.h>
#include <arm_fp16.h>

_Float16 goo(void)
{
   _Float16 a = 1.0f16;
   float16x4_t b = {1.0, 2.0, 3.0, 4.0};
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    a = vabsh_f16(a); // scalar half-precision floating-point intrinsic
    b = vabs_f16(b);  // vector half-precision floating-point intrinsic

    return a;
}
```

To compile the example for AArch64 state, use the command:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.2-a+fp16 -std=c90 -c foo.c -o
 foo.o
```

Note

Arm Compiler for Embedded 6 does not support the Armv8.2-A half-precision floating-point scalar intrinsics in AArch32 state.

If you want to use the Armv8.2-A half-precision floating-point scalar instructions in AArch32 state, you must either:

- Use the `_Float16` data type in your C or C++ source code.
- Use the `armclang` inline assembly or integrated assembler for instructions that cannot be generated from the source code.

Related information

-march on page 123
-mcpu on page 150
Using Assembly and Intrinsics in C or C++ Code
List of intrinsics
Arm C Language Extensions ACLE Q1 2019

2.6.8 Library support for _Float16 data type

The C standard library in Arm® Compiler for Embedded 6 does not support the `_Float16` data type.

Note

Using `_Float16` with `-ffp-mode=full` is not supported.

If you want to use any of the functions from the C standard library on the `_Float16` data type, then you must manually cast the `_Float16` value to a single-precision, or double-precision value, and then use the appropriate library function.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Also, the library function `printf` does not have a string format specifier for the `_Float16` data type. Therefore an explicit cast is required for the `_Float16` data type. The following example casts the `_Float16` value to a `double` for use in the `printf` function.

```
// foo.c
#include <stdlib.h>
#include <stdio.h>

_Float16 foo(void)
{
    _Float16 n = 1.0f16;

    // Cast the _Float16 value n to a double because there is no string format
 specifier for half-precision floating-point values.
    printf ("Hello World %f \n", (double)n);

    return n;
}
```

To compile this example with `armclang`, use the command:

```
armclang --target=arm-arm-none-eabi -march=armv8.2-a+fp16 -std=c90 -c foo.c -o foo.o
```

The `printf` function does not automatically cast the `_Float16` value. If you do not manually cast the `_Float16` value, `armclang` produces the `-Wformat` diagnostic message.

```
warning: format specifies type 'double' but the argument has type '_Float16' [-
Wformat]
printf ("Hello World %f\n", n);
```

Related information

-march on page 123
-mcpu on page 150
-ffp-mode on page 62
List of intrinsics
Arm C Language Extensions ACLE Q1 2019

2.6.9 BFloat16 floating-point number format

Arm® Compiler for Embedded supports the floating-point `__bf16` type.

BFloat16 is a floating-point format that occupies 16 bits. It is supported by Armv8.2 and later Application profile architectures.



Note

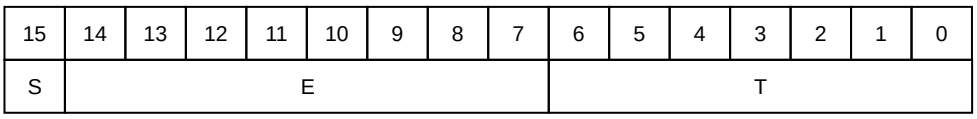
The `__bf16` type is a storage format only type, and it can only be used by intrinsics. An error is raised if arithmetic operations in C or C++ expressions are performed using the `__bf16` type.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

BFloat16 floating-point format

Arm Compiler for Embedded uses the BFloat16 binary floating-point format which is a truncated form of the IEEE 754 standard.

Figure 2-8: BFloat16 floating-point format



Where:

```
S (bit[15]):     Sign bit
E (bits[14:7]):  Biased exponent
T (bits[6:0]):   Fraction
```

Note

See the *Arm C Language Extensions* for more information.

Related information

Arm C Language Extensions ACLE Q1 2019

2.6.10 TT instruction intrinsics

Intrinsics are available to support `TT` instructions depending on the value of the predefined macro `__ARM_FEATURE_CMSE`.

TT intrinsics

The following table describes the `TT` intrinsics that are available when predefined macro `__ARM_FEATURE_CMSE` is set to either `1` or `3`:

Intrinsic	Description
`cmse_address_info_t cmse_TT(void *p)`	Generates a `TT` instruction.
`cmse_address_info_t cmse_TT_fptr(p)`	Generates a `TT` instruction. The argument `p` can be any function pointer type.
`cmse_address_info_t cmse_TTT(void *p)`	Generates a `TT` instruction with the `T` flag.
`cmse_address_info_t cmse_TTT_fptr(p)`	Generates a `TT` instruction with the `T` flag. The argument `p` can be any function pointer type.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

When predefined macro `__ARM_BIG_ENDIAN` is not set, the result of the intrinsics is returned in the following C type:

```
typedef union {
    struct cmse_address_info {
        unsigned mpu_region:8;
        unsigned :8;
        unsigned mpu_region_valid:1;
        unsigned :1;
        unsigned read_ok:1;
        unsigned readwrite_ok:1;
        unsigned :12;
    } flags;
    unsigned value;
} cmse_address_info_t;
```

When predefined macro `__ARM_BIG_ENDIAN` is set, the bit-fields in the type are reversed such that they have the same bit-offset as little-endian systems following the rules specified by *Procedure Call Standard for the Arm Architecture*.

TT intrinsics for Armv8-M Security Extension

The following table describes the `TT` intrinsics for Arm®v8-M Security Extension that are available when predefined macro `__ARM_FEATURE_CMSE` is set to `3`:

Intrinsic	Description
`cmse_address_info_t cmse_TTA(void *p)`	Generates a `TT` instruction with the `A` flag.
`cmse_address_info_t cmse_TTA_fptr(p)`	Generates a `TT` instruction with the `A` flag. The argument `p` can be any function pointer type.
`cmse_address_info_t cmse_TTAT(void *p)`	Generates a `TT` instruction with the `T` and `A` flag.
`cmse_address_info_t cmse_TTAT_fptr(p)`	Generates a `TT` instruction with the `T` and `A` flag. The argument `p` can be any function pointer type.

When the predefined macro `__ARM_BIG_ENDIAN` is not set, the result of the intrinsics is returned in the following C type:

```
typedef union {
    struct cmse_address_info {
        unsigned mpu_region:8;
        unsigned sau_region:8;
        unsigned mpu_region_valid:1;
        unsigned sau_region_valid:1;
        unsigned read_ok:1;
        unsigned readwrite_ok:1;
        unsigned nonsecure_read_ok:1;
        unsigned nonsecure_readwrite_ok:1;
        unsigned secure:1;
        unsigned idau_region_valid:1;
        unsigned idau_region:8;
    } flags;
    unsigned value;
} cmse_address_info_t;
```

When the predefined macro `__ARM_BIG_ENDIAN` is set, the bit-fields in the type are reversed such that they have the same bit-offset as little-endian systems following the rules specified by *Procedure Call Standard for the Arm Architecture*.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

In Secure state, the `TT` instruction returns the *Security Attribution Unit* (SAU) and *Implementation Defined Attribution Unit* (IDAU) configuration and recognizes the `A` flag.

Address range check intrinsic

Checking the result of the `TT` instruction on an address range is essential for programming in C. It is needed to check permissions on objects larger than a byte. You can use the address range check intrinsic to perform permission checks on C objects.

The syntax of this intrinsic is:

```
void *cmse_check_address_range(void *<p>, size_t <size>, int <flags>)
```

The intrinsic checks the address range from `<p>` to `<p> + <size> - 1`.

The address range check fails if `<p> + <size> - 1 < <p>`.

Some SAU, IDAU and MPU configurations block the efficient implementation of an address range check. This intrinsic operates under the assumption that the configuration of the SAU, IDAU, and MPU is constrained as follows:

- An object is allocated in a single region.
- A stack is allocated in a single region.

These points imply that a region does not overlap other regions.

The `TT` instruction returns an SAU, IDAU and MPU region number. When the region numbers of the start and end of the address range match, the complete range is contained in one SAU, IDAU, and MPU region. In this case two `TT` instructions are executed to check the address range.

Regions are aligned at 32-byte boundaries. If the address range fits in one 32-byte address line, a single `TT` instruction suffices. This is the case when the following constraint holds:

```
(<p> mod 32) + <size> <= 32
```

The address range check intrinsic fails if the range crosses any MPU region boundary.

The `<flags>` parameter of the address range check consists of a set of values defined by the macros shown in the following table:

Macro	Value	Description
(No macro)	0	The `TT` instruction without any flag is used to retrieve the permissions of an address, returned in a `cmse_address_info_t` structure.
`CMSE_MPU_UNPRIV`	4	Sets the `T` flag on the `TT` instruction used to retrieve the permissions of an address. Retrieves the unprivileged mode access permissions.
`CMSE_MPU_READWRITE`	1	Checks if the permissions have the `readwrite_ok` field set.
`CMSE_MPU_READ`	8	Checks if the permissions have the `read_ok` field set.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The address range check intrinsic returns `<p>` on a successful check, and `NULL` on a failed check. The check fails if any other value is returned that is not one of those listed in the table, or is not a combination of those listed.

We recommend that you use the returned pointer to access the checked memory range. This generates a data dependency between the checked memory and all its subsequent accesses and prevents these accesses from being scheduled before the check.

The following intrinsic is defined when the predefined macro `__ARM_FEATURE_CMSE` macro is set to `1`:

Intrinsic	Description
`cmse_check_pointed_object(<p>, <f>)`	Returns the same value as: `cmse_check_address_range(<p>, sizeof(*<p>), <f>)`

We recommend that the return type of this intrinsic is identical to the type of parameter `<p>`.

Address range check intrinsic for Armv8-M Security Extension

The semantics of the intrinsic `cmse_check_address_range()` are extended to handle the extra flag and fields introduced by the Armv8-M Security Extension.

The address range check fails if the range crosses any SAU or IDAU region boundary.

If the predefined macro `__ARM_FEATURE_CMSE` is set to `3`, the values accepted by the `<flags>` parameter are extended with the values defined in the following table:

Macro	Value	Description
`CMSE_AU_NONSECURE`	`2`	Checks if the permissions have the `secure` field unset.
`CMSE_MPU_NONSECURE`	`16`	Sets the `A` flag on the `TT` instruction used to retrieve the permissions of an address.
`CMSE_NONSECURE`	`18`	Combination of `CMSE_AU_NONSECURE` and `CMSE_MPU_NONSECURE`.

Related information

Predefined macros on page 336

2.6.11 Non-secure function pointer intrinsics

A non-secure function pointer is a function pointer that has its LSB unset.

The following table describes the non-secure function pointer intrinsics that are available when the predefined macro `__ARM_FEATURE_CMSE` is set to `3`:

Table 2-53: Non-secure function pointer intrinsics

Intrinsic	Description
`cmse_nsfptr_create(p)`	Returns the value of `p` with its LSB cleared. The argument `p` can be any function pointer type. We recommend that the return type of this intrinsic is identical to the type of its argument.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Intrinsic	Description
`cmse_is_nsfptr(p)`	Returns non-zero if `p` has LSB unset, zero otherwise. The argument `p` can be any function pointer type.

Example: Using the cmse_nsfptr_create and cmse_is_nsfptr intrinsics

The following example shows how to use these intrinsics:

```
#include <arm_cmse.h>

typedef void __attribute__((cmse_nonsecure_call)) nsfunc(void);
void default_callback(void) { ... }

// fp can point to a secure function or a non-secure function
nsfunc *fp = (nsfunc *) default_callback;           // secure function pointer

void __attribute__((cmse_nonsecure_entry)) entry(nsfunc *callback) {
    fp = cmse_nsfptr_create(callback);  // non-secure function pointer
}

void call_callback(void) {
    if (cmse_is_nsfptr(fp)) fp();       // non-secure function call
    else ((void (*)(void)) fp)();       // normal function call
}
```

Related information

__attribute__((cmse_nonsecure_call)) function attribute on page 260
__attribute__((cmse_nonsecure_entry)) function attribute on page 261
Predefined macros on page 336
Overview of building Secure and Non-secure images with the Armv8-M Security Extension

2.6.12 Supported architecture features

You specify the architecture features supported by Arm® Compiler for Embedded using either the `armclang` options `-march` or `-mcpu`.

2.6.12.1 A-profile Memory Operations Extension

The A-profile Memory Operations Extension adds support for the Arm®v8.8-A and Armv9.3-A instructions for the `memcpy`, `memmove`, and `memset` operations.

The supported instructions are:

Table 2-54: A-profile Memory Operations Extension

Operation	Instructions
`memcpy`	`CPYFP`, `CPYFM`, and `CPYFE`
`memmove`	`CPYP`, `CPYM`, and `CPYE`
`memset`	`SETP`, `SETM`, and `SETE`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

See Arm A-profile A64 -- Base Instructions (alphabetic order) for more information on these instructions.

2.6.12.2 Custom Datapath Extension

The *Custom Datapath Extension* (CDE) changes the *Instruction Set Architecture* (ISA) of the encoding space to CDEv1 for the corresponding coprocessor, cp0 to cp7.

This extension is only compatible with M-profile targets and requires the target to support at least Arm®v8-M.mainline.

Use the `+cdecp<N>` option, where `<N>` is in the range 0-7.

Related information

-march on page 123
-mcpu on page 150
Arm C Language Extensions

2.6.12.3 Cryptographic extensions

Arm® Compiler for Embedded supports various algorithms for the cryptographic extensions.

Note

Some Arm processors have cryptographic extensions enabled by default, and others do not. Therefore, we recommend that you explicitly specify the required cryptographic extensions.

Your processor Technical Reference Manual might use the term cryptographic extension to mean both scalar cryptographic extension features and *Scalable Vector Extension* (SVE) cryptographic extension features. SVE cryptographic features are not enabled using the `+crypto` feature modifier, and you must specify them explicitly. For information about SVE forms of cryptographic features, see Scalable Vector Extension.

The following tables show which algorithms are supported in different versions of the Armv8-A and Armv9-A architectures, and the Cortex®-R82 processor:

Table 2-55: Cryptographic extensions for AArch32 state

Feature	Armv8.0-A and later	Armv8-R	Armv9.0-A and later
`+aes`	AES, SHA1, SHA256	AES	AES

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 2-56: Cryptographic extensions for AArch64 state

Feature	Armv8.0-A, Armv8.1-A, Armv8.2-A, Armv8.3-A, Armv8-R, Armv8-R AArch64	Armv8.4-A and later, Armv9.0-A and later
`+crypto`	SHA1, SHA256, AES	SHA1, SHA256, SHA512, SHA3, AES, SM3, SM4
`+aes`	AES	AES
`+sha2`	SHA1, SHA256	SHA1, SHA256
`+sha3`	SHA1, SHA256, SHA512, SHA3	SHA1, SHA256, SHA512, SHA3
`+sm4`	SM3, SM4	SM3, SM4


Note

Armv8.0-A refers to the generic Armv8-A architecture without any incremental architecture extensions. On the `armclang` command-line, use `-march=armv8-a` to compile for Armv8.0-A.

Armv9.0-A refers to the generic Armv9-A architecture without any incremental architecture extensions. On the `armclang` command-line, use `-march=armv9-a` to compile for Armv9.0-A.

For AArch32 state in Armv8-A, Armv8-R, and Armv9-A, if you specify an `-mfpu` option that includes the Cryptographic extension, then the Cryptographic extension supports the AES, SHA1, and SHA256 algorithms.

Related information

-march on page 123
-mcpu on page 150
-mfpu on page 170

2.6.12.4 Floating-point extensions

Arm® Compiler for Embedded supports various floating-point features.

The following tables show the A-profile and R-profile floating-point instructions that are available when you use the floating-point features:


Note

For completeness, the tables also show the floating-point features that support the I8MM instructions for 8-bit integer operations. See Matrix Multiplication extension for more information.

For M-profile, see M-profile Vector Extension and Supported architecture feature combinations for specific processors for more information.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 2-57: Floating-point extensions for AArch32 state

Feature	Armv8.0-A, Armv8.1-A, Armv8.2-A, Armv8.3-A	Armv8.4-A	Armv8.5-A, Armv9.0-A	Armv8.6-A and later, Armv9.1-A and later	Armv8-R
`+fp`	SIMD, FP	SIMD, FP	SIMD, FP	SIMD, FP, BF16, I8MM	SIMD, FP
`+fp16`	SIMD, FP, FP16	SIMD, FP, FP16, FP16fml	SIMD, FP, FP16	SIMD, FP, FP16, BF16, I8MM	SIMD, FP, FP16
`+fp16fml`	SIMD, FP, FP16, FP16fml	SIMD, FP, FP16, FP16fml	SIMD, FP, FP16, FP16fml	SIMD, FP, FP16, FP16fml, BF16, I8MM	SIMD, FP, FP16, FP16fml
`+bf16`	SIMD, FP, BF16	SIMD, FP, BF16	SIMD, FP, BF16	SIMD, FP, BF16, I8MM	SIMD, FP, BF16

Table 2-58: Floating-point extensions for AArch64 state

Feature	Armv8.0-A, Armv8.1-A, Armv8.2-A, Armv8.3-A	Armv8.4-A	Armv8.5-A	Armv8.6-A and later	Armv8-R	Armv8-R AArch64	Armv9.0-A	Armv9.1-A, and later
`+fp`	FP	FP	FP	FP, BF16, I8MM	FP	FP, FP16, FP16fml	FP, FP16	FP, FP16, BF16, I8MM
`+fp16`	FP, FP16	FP, FP16, FP16fml	FP, FP16	FP, FP16, FP16fml	FP, FP16	FP, FP16, FP16fml	FP, FP16	FP, FP16, BF16, I8MM
`+fp16fml`	FP, FP16, FP16fml	FP, FP16, FP16fml	FP, FP16, FP16fml	FP, FP16, FP16fml	FP, FP16, FP16fml	FP, FP16, FP16fml	FP, FP16, FP16fml	FP, FP16, FP16fml, BF16, I8MM
`+bf16`	FP, BF16	FP, BF16	FP, BF16	FP, BF16	FP, BF16	FP, FP16, FP16fml, BF16	FP, FP16, BF16	FP, FP16, BF16, I8MM

FP refers to the single-precision and double-precision arithmetic operations.

FP16 refers to the Armv8.2-A half-precision floating-point arithmetic operations.

FP16fml refers to the half-precision floating-point multiply with add or multiply with subtract arithmetic operations. These operations are supported in the Armv8.2 and later application profile architectures, and are optional in Armv8.2-A and Armv8.3-A.



Note

Arm Compiler for Embedded 6 does not support the flushing of denormal half-precision values.

BF16 refers to the BFloat16 floating-point dot product, matrix multiplication, and conversion operations.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Armv8.0-A refers to the generic Armv8-A architecture without any incremental architecture extensions. On the `armclang` command-line, use `-march=armv8-a` to compile for Armv8.0-A.

Armv9.0-A refers to the generic Armv9-A architecture without any incremental architecture extensions. On the `armclang` command-line, use `-march=armv9-a` to compile for Armv9.0-A.


Note

Armv8-A without hardware floating-point is not supported in AArch64 state.

Armv8-R AArch64 state with hardware and software floating-point support is as follows:

- Armv8-R AArch64 state implementations with hardware floating-point are fully supported.
- Armv8-R AArch64 state implementations without hardware floating-point are only supported in C and assembly source language modes. C++ source language modes are not supported for such targets.

Related information

-march on page 123
-mcpu on page 150

2.6.12.5 Matrix Multiplication extension

Matrix Multiplication extension is a component of the Arm®v8.6-A architecture and is an optional extension for the Armv8.2-A to Armv8.5-A architectures.

The following table shows the features to enable the Matrix Multiplication extension:

Table 2-59: Features for the Matrix Multiplication extension

Feature	Description
`+i8mm`	Enables matrix multiplication instructions for 8-bit integer operations. This feature also enables the `+simd` feature in AArch64 state, or Advanced SIMD in AArch32 state.
`+f32mm`	Enables matrix multiplication instructions for 32-bit single-precision floating-point operations. This feature also enables the `+sve` feature in AArch64 state.
`+f64mm`	Enables matrix multiplication instructions for 64-bit double-precision floating-point operations. This feature also enables the `+sve` feature in AArch64 state.
`+f8f16mm`	Enables matrix multiplication instructions for Advanced SIMD FP8 to half-precision floating-point operations. This feature also enables the `+sve` and `+sve2` features at Armv9.0-A and later in AArch64 state.
`+f8f32mm`	Enables matrix multiplication instructions for Advanced SIMD FP8 to single-precision floating-point operations. This feature also enables the `+sve` and `+sve2` features at Armv9.0-A and later in AArch64 state.

The following tables show the instructions that are available for each feature of the Matrix Multiplication extension:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

For completeness, the tables also show the Matrix Multiplication features that support the floating-point instructions. See Floating-point extensions for more information.

Table 2-60: Matrix Multiplication extension for AArch32 state

Feature	Armv8.0-A and later	Armv8-R	Armv9.0-A and later
`+i8mm`	SIMD, FP, I8MM	SIMD, FP, I8MM	SIMD, FP, I8MM

Table 2-61: Matrix Multiplication extension for AArch64 state

Feature	Armv8.0-A, Armv8.1-A, Armv8.2-A, Armv8.3-A, Armv8.4-A, Armv8.5-A	Armv8.6-A and later	Armv8-R	Armv8-R AArch64	Armv9.0-A	Armv9.1-A, and later
`+i8mm`	FP, I8MM	FP, BF16, I8MM	FP, I8MM	FP, FP16, FP16fml, I8MM	FP, FP16, I8MM	FP, FP16, BF16, I8MM
`+f32mm`	FP, FP16, F32MM	FP, FP16, I8MM, F32MM	FP, FP16, F32MM	FP, FP16, FP16fml, F32MM	FP, FP16, F32MM	FP, FP16, BF16, I8MM, F32MM
`+f64mm`	FP, FP16, F64MM	FP, FP16, BF16, I8MM, F64MM	FP, FP16, F64MM	FP, FP16, FP16fml, F64MM	FP, FP16, F64MM	FP, FP16, BF16, I8MM, F64MM
`+f8f16mm`	FP, FP8, BF16, F8F16MM, F8F32MM	FP, FP8, BF16, I8MM, F8F16MM, F8F32MM	-	FP, FP8, FP16, BF16, F8F16MM, F8F32MM	FP, FP8, FP16, BF16, F8F16MM, F8F32MM	FP, FP8, FP16, BF16, I8MM, F8F16MM, F8F32MM
`+f8f32mm`	FP, FP8, BF16, F8F32MM	FP, FP8, BF16, I8MM, F8F32MM	-	FP, FP8, FP16, BF16, F8F32MM	FP, FP8, FP16, BF16, F8F32MM	FP, FP8, FP16, BF16, I8MM, F8F32MM

Arm Compiler for Embedded enables:

- Assembly of source code containing Matrix Multiplication instructions.
- Disassembly of ELF object files containing Matrix Multiplication instructions.
- Support for the ACLE defined Matrix Multiplication intrinsics.

Related information

-march on page 123
-mcpu on page 150

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.6.12.6 M-profile Vector Extension

M-profile Vector Extension (MVE) is an optional extension for the Arm®v8.1-M architecture profile.



Note

For some Arm processors, `-mcpu` supports specific combinations of this architecture feature. See Supported architecture feature combinations for specific processors for more information.

The following table shows the features to enable MVE:

Table 2-62: Features for the MVE extension

Feature	Description
`+mve`	Enables MVE instructions for integer operations.
`+mve.fp`	Enables MVE instructions for integer, half-precision floating-point, and single-precision floating-point operations. If you use the `double` data type, the compiler generates library calls for double-precision floating-point operations. Using library calls can result in extra code size and lower performance. If the processor supports scalar double-precision floating-point instructions, consider using `+mve.fp+fp.dp`.
`+mve.fp+fp.dp`	Enables MVE instructions for integer, half-precision floating-point, and single-precision floating-point operations. With `+fp.dp`, using the `double` data type causes the compiler to generate scalar double-precision floating-point instructions. If the processor does not support scalar double-precision floating-point instructions, then the processor faults. In that case, use `+mve.fp` without `+fp.dp`. **Note:** For the Cortex®-M55 processor, `+mve.fp+fp.dp` is unsupported. Use `mcpu=cortex-m55` instead. See Supported architecture feature combinations for specific processors for more information.

Arm Compiler for Embedded enables:

- Assembly of source code containing MVE instructions.
- Disassembly of ELF object files containing MVE instructions.
- Support for the ACLE defined MVE intrinsics.
- Automatic vectorization of source code operations into MVE instructions.

Related information

-march on page 123
-mcpu on page 150
Arm C Language Extensions

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.6.12.7 Armv8.1-M PACBTI extension

The Arm®v8.1-M PACBTI extension is an optional extension for the Armv8.1-M architecture profile.

PACBTI affects the choice of instructions that are emitted when branch protection is enabled using the `-mbranch-protection` option. If the extension is enabled, the compiler is allowed to use non-hint-space instructions. Otherwise, the compiler uses only hint-space instructions.


Note

The `+pacbti` feature is required to use the PACBTI registers in `MSR` and `MRS` instructions, and PACBTI instructions in handwritten assembly code.

The Armv8.1-M PACBTI extension consists of the implementation of the following control-flow integrity approaches:

- Return address signing and authentication (PAC-RET) as a mitigation for *Return Oriented Programming* (ROP) style attack.
- `BTI` instruction placement (BTI) as a mitigation for *Jump Oriented Programming* (JOP) style attacks.

Without `+pacbti`, the compiler always uses hint-space instructions. That is, the compiler does not use the `BXAUT` instruction.

For more information, see:

- Armv8-M Architecture Reference Manual
- Learn the architecture: Providing protection for complex software

Related information

-march on page 123
-mcpu on page 150
Armv8-M Architecture Reference Manual
Learn the architecture: Providing protection for complex software

2.6.12.8 Scalable Matrix Extension

Scalable Matrix Extension (SME) introduces A64 instructions for the Arm®v9-A application profile architecture.

SME also introduces the ZA array. The ZA array is a new architectural register state consisting of a matrix of [SVLb x SVLb] bytes, where SVL is the implementation defined Streaming SVE vector length and SVLb is the number of 8-bit elements in a vector of SVL bits.

Arm Compiler for Embedded enables:

- Assembly of source code containing SME instructions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Disassembly of ELF object files containing SME instructions.

Table 2-63: Options for the SME extensions

Feature	Description
`+sme`	Enables SME instructions (FEAT_SME) Also enables: • Streaming SVE, which is a significant subset of SVE2. • bf16.
`+sme2`	Extends `+sme` to support SME2 instructions (FEAT_SME2). For Armv9.4-A, also enables `+sme2p1` (FEAT_SME2p1).
`+sme-f64f64`	Enables SME instructions with the double-precision variant (FEAT_SME_F64F64).
`+sme-i16i64`	Enables SME instructions with the 16-bit integer variant (FEAT_SME_I16I64).
`+sme2p1`	Extends `+sme2` to support SME2.1 instructions (FEAT_SME2p1). `+sme2p1` extends `+sve2p1`.
`+sve-b16b16`	Extends `+sme2p1` with non-widening BFloat16 instructions (FEAT_SVE_B16B16).
`+sme-b16b16`	Non-widening BFloat16 instructions (FEAT_SME_B16B16).
`+sme-f16f16`	Enables SME instructions with the 16-bit integer variant (FEAT_SME_F16F16).
`+sme2p2`	Extends `+sme2` to support SME2.2 instructions (FEAT_SME2p2).

See A64 - SME Instructions (alphabetic order) for more information.

Related information

-march on page 123
-mcpu on page 150
Arm C Language Extensions

2.6.12.9 Scalable Vector Extension

Scalable Vector Extension (SVE) is a SIMD instruction set for Arm®v8-A AArch64.

SVE also introduces the following architectural features:

- Scalable vector length.
- Per-lane predication.
- Gather-load and scatter-store.
- Fault-tolerant Speculative vectorization.
- Horizontal and serialized vector operations.

Arm Compiler for Embedded enables:

- Assembly of source code containing SVE instructions.
- Disassembly of ELF object files containing SVE instructions.
- Intrinsics.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Auto-vectorization.

Table 2-64: SVE options

Feature	Description
`+sve`	Enables SVE instructions (FEAT_SVE), `+fp16` (FEAT_FP16), and `+fp16fml` (FEAT_FHM).
`+sve2`	SVE2 builds on SVE to add many new data-processing instructions that bring the benefits of scalable long vectors to a wider class of applications (FEAT_SVE2). For Armv9.4-A, also enables `+sve2p1` (FEAT_SVE2p1). To enable only the base SVE2 instructions, use the `+sve2` option.
`+sve2-aes`	Enables SVE forms of the cryptographic instructions: • `AESD`, `AESE`, `AESIMC`, and `AESMC` (FEAT_SVE_AES). • `PMULLB` and `PMULLT` instructions (FEAT_SVE_PMULL128).
`+sve2-bitperm`	Enables the `BDEP`, `BEXT`, and `BGRP` instructions (FEAT_SVE_BitPerm).
`+sve2-sha3`	Enables SVE forms of the `RAX1` cryptographic instruction (FEAT_SVE_SHA3).
`+sve2-sm4`	Enables SVE forms of `SM4E` and `SM4EKEY` cryptographic instructions (FEAT_SVE_SM4).
`+sve2p1`	Extends `+sve2` to support a number of instructions that have been moved from SME, and includes changes to Streaming SVE mode instructions in SVE2.1 (FEAT_SVE2p1).
`+sve-b16b16`	Extends `+sve2p1` with non-widening BFloat16 instructions (FEAT_SVE_B16B16).
`+sve2p2`	Enables SVE2.2 instructions in Non-streaming SVE mode (FEAT_SVE2p2).

You can use one or more of these options. Except for `+sve`, each option also implies `+sve2`. For example, `+sve2-aes+sve2-bitperm+sve2-sha3+sve2-sm4` enables all base and optional instructions. For clarity, you can include `+sve2` if necessary.

See Arm A-profile A64 -- SVE Instructions (alphabetic order) for more information.

Related information

-march on page 123
-mcpu on page 150
Arm C Language Extensions

2.6.12.10 Transactional Memory Extension

Transactional Memory Extension (TME) is an architecture extension that adds instructions to support lock-free atomic execution of critical sections.



Note

Arm® Compiler for Embedded does not support single-copy atomic instructions for accessing `volatile` variables that are larger than 32 bits for AArch32 and 64 bits for AArch64.

Related information

-march on page 123

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

-mcpu on page 150
Arm C Language Extensions

2.6.12.11 Prevention of Speculative execution and data prediction

A processing element in an Arm®v8-A device can perform Speculative execution of instructions within a particular execution context by default.

At any point in time, a *Processing Element* (PE) knows about instructions that are executed at an earlier point in time. The PE uses this knowledge to make predictions to perform Speculative execution of instructions at a later point in time.

This behavior can be mitigated by using special instructions that prevent these predictions. When such an instruction is executed, it prevents the PE from using certain knowledge of the instructions executed before it. Therefore, the PE cannot perform certain Speculative execution of instructions executed after it.

Support for `+predres` is optional for architectures Armv8.0-A to Armv8.4-A, and mandatory for architecture Armv8.5-A and later. The following features enable these instructions:

- `+predres` is available in AArch64 state, and enables the instructions:

```
cfp RCTX, Xt
dvp RCTX, Xt
cpp RCTX, Xt
```

- `+sb` is available in AArch32 and AArch64 states, and enables the `SB` instruction.
- `+ssbs` is available in AArch64 state, and enables the instructions:

```
mrs Xt, SSBS
msr SSBS, Xt
```

The Enhanced Speculation Management feature includes an additional speculation restriction instruction. This instruction clears the influence of any trained speculation mechanisms other than those managed by the existing `cfp`, `cpp`, and `dvp` instructions.

Support for the Enhanced Speculation Management feature is mandatory for architecture Armv8.9-A and later.

`+predres2` is available in AArch64 state, and enables the instruction:

```
cosp RCTX, Xn
```

Related information

-march on page 123
-mcpu on page 150

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.6.13 Supported architecture feature combinations for specific processors

For some Arm processors, the `armclang` option `-mcpu` and the `armlink` and `fromelf` option `--cpu` support specific combinations of the architecture features.

For `armclang`, the options in the tables assume that you also include `--target=arm-arm-none-eabi`.


Note

The default options in an *Integrated Development Environment* (IDE) might be different and might override the default options for the toolchain.

If you are building a validation test provided as part of the IP deliverables for your processor, see the Release Notes and makefiles included in those deliverables for details of the command-line options being used.

Combinations of architecture features supported for the Cortex-M52 processor

The following *M-profile Vector Extension* (MVE), *Floating-point* (FP), and PACBTI combinations for the Cortex®-M52 processor are supported:


Note

Do not use the `armlink` option `--cpu` when linking.

Table 2-65: Combinations of architecture features supported for the Cortex-M52 processor

Scalar FP half-precision and single-precision	Scalar FP double-precision	MVE integer	MVE FP half-precision and single-precision	M-profile PACBTI Extension[1]	`armclang` option `-mcpu`	`armlink` option `--cpu`	`fromelf` option `--cpu`[2]
Included	Included	Included	Included	Included	`cortex-m52`	-	`Armv8.1-M.Main.mve`
Included	Included	Not included	Not included	Included	`cortex-m52+nomve`	-	`Armv8.1-M.Main.mve`
Included	Not included	Included	Not included	Included	`cortex-m52+nomve.fp+nofp.dp`	-	`Armv8.1-M.Main.mve`
Included	Not included	Not included	Not included	Included	`cortex-m52+nomve+nofp.dp`	-	`Armv8.1-M.Main.mve`
Not included	Not included	Included	Not included	Included	`cortex-m52+nomve.fp+nofp`	-	`Armv8.1-M.Main.mve`
Included	Included	Included	Included	Not included	`cortex-m52+nopacbti`	-	`Armv8.1-M.Main.mve`
Included	Included	Not included	Not included	Not included	`cortex-m52+nomve+nopacbti`	-	`Armv8.1-M.Main.mve`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Scalar FP half-precision and single-precision	Scalar FP double-precision	MVE integer	MVE FP half-precision and single-precision	M-profile PACBTI Extension[1]	armclang option –mcpu	armlink option --cpu	fromelf option --cpu[2]
Included	Not included	Included	Not included	Not included	cortex-m52+nomve.fp+nofp.dp+nopacbti	-	Armv8.1-M.Main.mve
Included	Not included	Not included	Not included	Not included	cortex-m52+nomve+nofp.dp+nopacbti	-	Armv8.1-M.Main.mve
Not included	Not included	Included	Not included	Not included	cortex-m52+nomve.fp+nofp+nopacbti	-	Armv8.1-M.Main.mve

Table notes

[1] Although the M-profile PACBTI Extension is enabled by default, `armclang` does not automatically insert PACBTI instructions into user code by default. You must also use the `armclang` option `-mbranch-protection` to generate the PACBTI instructions. Also, the M-profile PACBTI variant of the Arm C libraries is not selected by default. For more information, see the -mbranch-protection and --library_security=protection.

[2] The `--cpu=Armv8.1-M.Main.mve` option for the ELF processing utility `fromelf` is required to enable successful disassembly of all Arm®v8.1-M and MVE instructions.

Combinations of architecture features supported for the Cortex-M55 processor

The following MVE and FP combinations for the Cortex-M55 processor are supported:

Table 2-66: Combinations of architecture features supported for the Cortex-M55 processor

Scalar FP half-precision and single-precision	Scalar FP double-precision	MVE integer	MVE FP half-precision and single-precision	armclang option –mcpu	armlink and fromelf option --cpu
Included	Included	Included	Included	cortex-m55	Cortex-M55
Included	Included	Included	Not included	cortex-m55+nomve.fp	Cortex-M55.no_mvefp
Not included	Not included	Included	Not included	cortex-m55+nofp	Cortex-M55.no_fp
Included	Included	Not included	Not included	cortex-m55+nomve	Cortex-M55.no_mve
Not included	Not included	Not included	Not included	cortex-m55+nofp+nomve	Cortex-M55.no_mve.no_fp

Combinations of architecture features supported for the Cortex-M85 processor

The following MVE, FP, and PACBTI combinations for the Cortex-M85 processor are supported:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 2-67: Combinations of architecture features supported for the Cortex-M85 processor

Scalar FP half-precision and single-precision	Scalar FP double-precision	MVE integer	MVE FP half-precision and single-precision	M-profile PACBTI Extension[1]	`armclang` option `-mcpu`	`armlink` and `fromelf` option `--cpu`
Included	Included	Included	Included	Included	`cortex-m85`	`Cortex-M85`
Not included	Not included	Included	Not included	Included	`cortex-m85+nofp`	`Cortex-M85.no_fp`
Included	Included	Not included	Not included	Included	`cortex-m85+nomve`	`Cortex-M85.no_mve`
Not included	Not included	Not included	Not included	Included	`cortex-m85+nofp+nomve`	`Cortex-M85.no_mve.no_fp`
Included	Included	Included	Included	Not included	`cortex-m85+nopacbti`	`Cortex-M85`
Not included	Not included	Included	Not included	Not included	`cortex-m85+nofp+nopacbti`	`Cortex-M85.no_fp`
Included	Included	Not included	Not included	Not included	`cortex-m85+nomve+nopacbti`	`Cortex-M85.no_mve`
Not included	Not included	Not included	Not included	Not included	`cortex-m85+nofp+nomve+nopacbti`	`Cortex-M85.no_mve.no_fp`

Table note

[1] Although the M-profile PACBTI Extension is enabled by default, `armclang` does not automatically insert PACBTI instructions into user code by default. You must also use the `armclang` option `-mbranch-protection` to generate the PACBTI instructions. Also, the M-profile PACBTI variant of the Arm C libraries is not selected by default. For more information, see the -mbranch-protection and --library_security=protection.

Related information

-mbranch-protection on page 144
-mcpu on page 150
--target on page 221
--cpu=list (armlink) on page 442
--library_security=protection on page 492
--cpu=list (fromelf) on page 779
Half-precision floating-point data types on page 342
Armv8.1-M PACBTI extension on page 360
Single precision data type for IEEE 754 arithmetic
Double precision data type for IEEE 754 arithmetic
Armv8.1-M PACBTI extension mitigations against ROP and JOP style attacks

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.7 armclang Integrated Assembler

The integrated assembler allows you to write code in the `armclang` integrated assembler syntax and assemble that code without having to use an external assembler.


Note

The integrated assembler sets a minimum alignment of 4 bytes for a `.text` section. However, if you define your own sections with the integrated assembler, then you must include the `.balign` directive to set the correct alignment. For a section containing T32 instructions, set the alignment to 2 bytes. For a section containing A32 instructions, set the alignment to 4 bytes.


Note

The integrated assembler incorrectly fails to report an error for a T32 instruction with an invalid `.n` width specifier. Instead, the integrated assembler assembles the instruction as a 32-bit instruction.

For example, the integrated assembler incorrectly fails to report an error for the following instruction:

```
    adc.n r0, r1, #1
```


Note

The integrated assembler might report an error if an instruction includes the `.w` width specifier.

For example, create `adc.s` containing the `ADC` instruction `adc.w r0, r1, 0x99`:

```
func:
  adc.w r0, r1, 0x99
```

Compiling the program gives an error:

```
armclang --target=arm-arm-none-eabi -march=armv7-m -o adc.o adc.s
adc.s:2:3: error: invalid instruction, any one of the following would
 fix this:
  adc.w r0, r1, 0x99
  ^
adc.s:2:17: note: invalid operand for instruction
  adc.w r0, r1, 0x99
                ^
adc.s:2:17: note: operand must be a register in range [r0, r12] or r14
  adc.w r0, r1, 0x99
                ^
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.7.1 Syntax of assembly files for integrated assembler

Assembly statements can include labels, instructions, directives, or macros.

Syntax

```
<label>:
    <instruction>[;]
    <directive>[;]
    <macro_invocation>[;]
```

Description

<label>

For label statements, the statement ends after the `:` character. For the other forms of assembler statements, the statement ends at the first newline or `;` character. This means that any number of labels can be defined on the same source line, and multiple of any other types of statements can be present in one source line if separated by `;`.

Label names without double quotes:

- Must start with a period (`.`), `_`, `a-z` or `A-Z`.
- Can also contain numbers, `_`, `$`.
- Must not contain white spaces.

You can have white spaces in label names by surrounding them with double quotes. Escape sequences are not interpreted within label names. It is also not possible to have double quotes as part of the label name.

<instruction>

Use the optional `;` to end the statement and start a new statement on the same line.

<directive>

Use the optional `;` to end the statement and start a new statement on the same line.

<macro_invocation>

Use the optional `;` to end the statement and start a new statement on the same line.

Comments

Comments are treated as equivalent to whitespace. The assembler ignores them.

There are three ways to include comments in an assembly file:

```
// single-line comment
@ single-line comment in AArch32 state only
/* multi-line
   comment */
```

In single-line comments, the `//` marker starts a comment that runs to the end of the source line. Unlike comments in C and C++ source, you cannot escape the end of the line with `\` to continue the comment.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`@` starts a single-line comment in AArch32 state. `@` is not a comment character in AArch64 state.

In multi-line comments, the `/*` marker starts a comment that runs to the first occurrence of `*/`, even if that is on a later line. Like comments in C and C++ source, the comment always ends at the first `*/`. Therefore, you cannot nest comments. You can use this style of comments anywhere within an assembly statement where whitespace is valid.


Note

Comments inside source files and header files that are provided by Arm might not be accurate and must not be treated as documentation about the product.

Examples

```
// Instruction on its own line:
      add r0, r1, r2

// Label and directive:
lab:  .word 42

// Multiple labels on one line:
lab1: lab2:

/* Multiple instructions, directives or macro-invocations
   must be separated by ';' */
      add r0, r1, r2; bx lr

// Multi-line comments can be used anywhere whitespace can:
      add /*dst*/r0, /*lhs*/r1, /*rhs*/r2
```

2.7.2 Assembly expressions

Expressions consist of one or more integer literals or symbol references, combined using operators.

You can use an expression when an instruction operand or directive argument expects an integer value or label.

Not all instruction operands and directive arguments accept all possible expressions. For example, the alignment directives require an absolute expression for the boundary to align to. Therefore, alignment directives cannot accept expressions involving labels, but can accept expressions involving only integer constants.

On the other hand, the data definition directives can accept a wider range of expressions, including references to defined or undefined symbols. However, the types of expressions accepted is still limited by the ELF relocations available to describe expressions involving undefined symbols. For example, it is not possible to describe the difference between two symbols defined in different sections. The assembler reports an error when an expression is not valid in the context in which it is used.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Expressions involving integer constants are evaluated as signed 64-bit values internally to the assembler. If an intermediate value in a calculation cannot be represented in 64 bits, the behavior is undefined. The assembler does not currently emit a diagnostic when this happens.

Constants

Numeric literals are accepted in the following formats:

- Decimal integer in range `0` to 2^{64}-1.
- Hexadecimal integer in range `0` to 2^{64}-1, prefixed with `0x`.
- Octal integer in range `0` to 2^{64}-1, prefixed with `0`.
- Binary integer in range `0` to 2^{64}-1, prefixed with `0b`.

Some directives accept values larger than 2^{64}-1. These directives only accept simple integer literals, not expressions.


Note

These ranges do not include negative numbers. Negative numbers can instead be represented using the unary operator, `-`.

Symbol References

References to symbols are accepted as expressions. Symbols do not need to be defined in the same assembly language source file, to be referenced in expressions.

The period symbol (`.`) is a special symbol that can be used to reference the current location in the output file.

For AArch32 targets, a symbol reference might optionally be followed by a modifier in parentheses. The following modifiers are supported:

Table 2-68: Modifiers

Modifier	Meaning
None	Do not relocate this value.
`got_prel`	Offset from this location to the GOT entry of the symbol.
`target1`	Defined by platform ABI.
`target2`	Defined by platform ABI.
`prel31`	Offset from this location to the symbol. Bit 31 is not modified.
`sbrel`	Offset to symbol from addressing origin of its output segment.
`got`	Address of the GOT entry for the symbol.
`gotoff`	Offset from the base of the GOT to the symbol.

Operators

The following operators are valid expressions:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 2-69: Unary operators

Unary operator	Meaning
-<expr>	Arithmetic negation of <expr>.
+<expr>	Arithmetic addition of <expr>.
~<expr>	Bitwise negation of <expr>.

Table 2-70: Binary operators

Binary operator	Meaning
<expr1> - <expr2>	Subtraction.
<expr1> + <expr2>	Addition.
<expr1> * <expr2>	Multiplication.
<expr1> / <expr2>	Division.
<expr1> % <expr2>	Modulo.

Table 2-71: Binary logical operators

Binary logical operator	Meaning
<expr1> && <expr2>	Logical and. 1 if both operands non-zero, 0 otherwise.
<expr1> || <expr2>	Logical or. 1 if either operand is non-zero, 0 otherwise.

Table 2-72: Binary bitwise operators

Binary bitwise operator	Meaning
<expr1> & <expr2>	<expr1> bitwise and <expr2>.
<expr1> | <expr2>	<expr1> bitwise or <expr2>.
<expr1> ^ <expr2>	<expr1> bitwise exclusive-or <expr2>.
<expr1> >> <expr2>	Logical shift right <expr1> by <expr2> bits.
<expr1> << <expr2>	Logical shift left <expr1> by <expr2> bits.

Table 2-73: Binary comparison operators

Binary comparison operator	Meaning
<expr1> == <expr2>	<expr1> equal to <expr2>.
<expr1> != <expr2>	<expr1> not equal to <expr2>.
<expr1> < <expr2>	<expr1> less than <expr2>.
<expr1> > <expr2>	<expr1> greater than <expr2>.
<expr1> <= <expr2>	<expr1> less than or equal to <expr2>.
<expr1> >= <expr2>	<expr1> greater than or equal to <expr2>.

The order of precedence for binary operators is as follows, with highest precedence operators listed first:

1. *, /, %, >>, <<
2. |, ^, &
3. +, -
4. ==, !=, <, >, <=, >=

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5. `&&`
6. `||`

Operators listed on the same line have equal precedence, and are evaluated from left to right. All unary operators have higher precedence than any binary operators.


Note

The precedence rules for assembler expressions are not identical to those for C.

Relocation specifiers for AArch32 and AArch64

For some instruction operands, a relocation specifier might be used to specify which bits of the expression to use for the operand, and which type of relocation to use.

These relocation specifiers can only be used at the start of an expression. They can only be used in operands of instructions that support them.

In AArch32 state, the following relocation specifiers are available:

Table 2-74: Relocation specifiers for AArch32 state

Relocation specifier	Meaning
`:lower16:`	Use the lower 16 bits of the expression value.
`:upper16:`	Use the upper 16 bits of the expression value.
`:lower0_7:`	Bits [0-7] of the lower 16 bits of the expression value.
`:lower8_15:`	Bits [8-15] of the lower 16 bits of the expression value.
`:upper0_7:`	Bits [0-7] of the upper 16 bits of the expression value.
`:upper8_15:`	Bits [8-15] of the upper 16 bits of the expression value.

The `:lower16:` and `:upper16:` relocation specifiers are only valid for the operands of the `movw` and `movt` instructions. They can be combined with an expression involving the current place to create a place-relative relocation, and with the `sbrel` symbol modifier to create a static-base-relative relocation. The current place is the location that the assembler is emitting code or data at. A place-relative relocation is a relocation that generates the offset from the relocated data to the symbol it references.

For Arm®v6-M a series of `movs`, `lsls`, and `adds` instructions is required to create a 32-bit expression value. The `:lower0_7:`, `:lower8_15:`, `:upper0_7:`, and `:upper8_15:` relocation specifiers are only valid for the operands of the `movs` and `adds` instructions.

In AArch64 state, the following relocation specifiers are available:

Table 2-75: Relocation specifiers for AArch64 state

Relocation specifier	Relocation type	Bits to use	Overflow checked
`:lo12:`	Absolute	[11:0]	No

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Relocation specifier	Relocation type	Bits to use	Overflow checked
:abs_g3:	Absolute	[63:48]	Yes
:abs_g2:	Absolute	[47:32]	Yes
:abs_g2_s:	Absolute, signed	[47:32]	Yes
:abs_g2_nc:	Absolute	[47:32]	No
:abs_g1:	Absolute	[31:16]	Yes
:abs_g1_s:	Absolute, signed	[31:16]	Yes
:abs_g1_nc:	Absolute	[31:16]	No
:abs_g0:	Absolute	[15:0]	Yes
:abs_g0_s:	Absolute, signed	[15:0]	Yes
:abs_g0_nc:	Absolute	[15:0]	No
:got:	Global Offset Table Entry	[32:12]	Yes
:got_lo12:	Global Offset Table Entry	[11:0]	No

These relocation specifiers can only be used in the operands of instructions that have matching relocations defined in ELF for the ARM 64-bit Architecture (AArch64). They can be combined with an expression involving the current place to create a place-relative relocation.

Example: Using an absolute expression in an instruction operand

```
    orr r0, r0, #1<<23
```

Example: Using an expression in the memory operand of an LDR instruction to reference an offset from a symbol

```
func:
    ldr r0, #data+4 // Loads the value 2 into r0
    bx lr
data:
    .word 1
    .word 2
```

Example: Creating initialized data that contains the distance between two labels

```
size:
    .word end - start
start:
    .word 123
    .word 42
    .word 4523534
end:
```

Example: Load the base-relative address of sym into r0 using movw and movt

This example is used for *Read/Write Position Independent* (RWPI) code:

```
    movw r0, #:lower16:sym(sbrel)
    movt r0, #:upper16:sym(sbrel)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Load the address of sym from the GOT using ADRP and LDR

This example is used for position independent code on AArch64:

```
    adrp x0, #:got:sym
    ldr x0, [x0, #:got_lo12:sym]
```

Example: Constant pool entry containing the offset between the location and a symbol defined elsewhere

The address of the symbol can be calculated at runtime by adding the value stored in the location of the address of the location. This method is one technique for writing position independent code, which can be executed from an address chosen at runtime without re-linking it.

```
    adr r0, address
    ldr r1, [r0]
    add r0, r0, r1
address:
    .word extern_symbol - .
```

Example: Create a 32-bit constant on an Armv6-M target

```
    movs    r3, #:upper8_15:#.LC0
    lsls    r3, #8
    adds    r3, #:upper0_7:#.LC0
    lsls    r3, #8
    adds    r3, #:lower8_15:#.LC0
    lsls    r3, #8
    adds    r3, #:lower0_7:#.LC0
    ldr     r3, [r3]
```

2.7.3 Alignment directives

The alignment directives align the current location in the file to a specified boundary.



The integrated assembler sets a minimum alignment of 4 bytes for a `.text` section. However, if you define your own sections with the integrated assembler, then you must include the `.balign` directive to set the correct alignment. For a section containing T32 instructions, set the alignment to 2 bytes. For a section containing A32 instructions, set the alignment to 4 bytes.

Syntax

```
.balign <num_bytes> [, <fill_value>]

.balignl <num_bytes> [, <fill_value>]

.balignw <num_bytes> [, <fill_value>]

.p2align <exponent> [, <fill_value>]

.p2alignl <exponent> [, <fill_value>]

.p2alignw <exponent> [, <fill_value>]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
.align <exponent> [, <fill_value>]
```

Parameters

<num_bytes>

This parameter specifies the number of bytes that must be aligned to. The number must be a power of 2.

<exponent>

This parameter specifies the alignment boundary as an exponent. The actual alignment boundary is $2^{<exponent>}$.

<fill_value>

The value to fill any inserted padding bytes with. This value is optional. The `w` and `l` suffixes modify the width of the padding value that is inserted:

- By default, the `<fill_value>` is a 1-byte value.
- The `w` suffix specifies that the `<fill_value>` is a 2-byte value.
- The `l` suffix specifies that the `<fill_value>` is a 4-byte value.

Operation

Use the alignment directives to ensure that your data and code are aligned to appropriate boundaries. Alignment is typically required in the following circumstances:

- In T32 code, the ADR instruction and the PC-relative version of the LDR instruction can only reference addresses that are 4-byte aligned, but a label within T32 code might only be 2-byte aligned. Use `.balign 4` to ensure 4-byte alignment of an address within T32 code.
- Use alignment directives to take advantage of caches on some Arm processors. For example, many processors have an instruction cache with 16-byte lines. Use `.p2align 4` or `.balign 16` to align function entry points on 16-byte boundaries to maximize the efficiency of the cache.


Note

In both of the following examples, it is important that the directive comes before the label that is to be aligned. If the label came first, then it would point at the padding bytes, and not the function or data it is intended to point to.

The alignment directives align the current location in the file to a specified boundary. The unused space between the previous and the new current location are filled with:

- Copies of `<fill_value>`, if it is specified. You can control the width of `<fill_value>` with the `w` and `l` suffixes.
- NOP instructions appropriate to the current instruction set, if all the following conditions are specified:
  - The `<fill_value>` argument is not specified.
  - The `w` or `l` suffix is not specified.
  - The alignment directive follows an instruction.
- Zeroes otherwise.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The `.balign` directive takes an absolute number of bytes as its first argument, and the `.p2align` directive takes a power of 2. For example, the following directives align the current location to the next multiple of 16 bytes:

- `.balign 16`
- `.p2align 4`
- `.align 4`

If you specify either the `w` or `l` suffix, the padding values are emitted as data (defaulting to a value of zero), even if following an instruction.

The `.align` directive is an alias for `.p2align`, but it does not accept the `w` and `l` suffixes.

Alignment is relative to the start of the section in which the directive occurs. If the current alignment of the section is lower than the alignment requested by the directive, the alignment of the section is increased.

Example: Align a constant pool value to a 4-byte boundary in T32 code

```
get_val:
  ldr r0, value
  adds r0, #1
  bx lr
  // This code is 6 bytes in size.
  // Therefore, the data defined by the following .word directive
  // must be manually aligned to a 4-byte boundary to be used by
  // the LDR instruction.
  .p2align 2
value:
  .word 42
```

Example: Ensure that the entry points to functions are on 16-byte boundaries, to better utilize caches

```
  .p2align 4
  .type func1, "function"
func1:
  // code

  .p2align 4
  .type func2, "function"
func2:
  // code
```

2.7.4 Data definition directives

These directives allocate memory in the current section, and define the initial contents of that memory.

Syntax

```
.byte <expr>[, <expr>]...

.hword <expr>[, <expr>]...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
.word <expr>[, <expr>]...

.quad <expr>[, <expr>]...

.octa <expr>[, <expr>]...
```

Parameters

<expr>

An expression that has one of the following forms:

- A absolute value, or expression (not involving labels) which evaluates to one. For example:

```
    .word (1<<17) | (1<<6)
    .word 42
```

- An expression involving one label, which might or might not be defined in the current file, plus an optional constant offset. For example:

```
    .word label
    .word label + 0x18
```

- A place-relative expression, involving the current location in the file (or a label in the current section) subtracted from a label which might either be defined in another section in the file, or undefined in the file. For example:

```
foo:
    .word label - .
    .word label - foo
```

- A difference between two labels, both of which are defined in the same section in the file. The section containing the labels need not be the same as the one containing the directive. For example:

```
    .word end - start
start:
        // ...
end:
```

Operation

The number of bytes allocated by each directive is as follows:

Table 2-76: Data definition directives

Directive	Size in bytes
.byte	1
.hword	2
.word	4
.quad	8
.octa	16

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If multiple arguments are specified, multiple memory locations of the specified size are allocated and initialized to the provided values in order.

The following table shows which expression types are accepted for each directive. In some cases, this varies between AArch32 and AArch64. This is because the two architectures have different relocation codes available to describe expressions involving symbols defined elsewhere. For absolute expressions, the table gives the range of values that are accepted (inclusive on both ends).

Table 2-77: Expression types supported by the data definition directives

Directive	Absolute	Label	Place-relative	Difference
`.byte`	Within the range `[-128,255]` only	AArch32 only	Not supported	AArch64 and AArch32
`.hword`	Within the range `[-0x8000,0xffff]` only	AArch64 and AArch32	AArch64 only	AArch64 and AArch32
`.word`	Within the range `[-2^31,2^32-1]` only	AArch64 and AArch32	AArch64 and AArch32	AArch64 and AArch32
`.quad`	Within the range `[-2^63,2^64-1]` only	AArch64 only	AArch64 only	AArch64 only
`.octa`	Within the range `[0,2^128-1]` only	Not supported	Not supported	Not supported



Note

While most directives accept expressions, the `.octa` directive only accepts literal values. In the `armclang` inline assembler and integrated assembler, negative values are expressions (the unary negation operator and a positive integer literal), so negative values are not accepted by the `.octa` directive. If negative 16-byte values are needed, you can rewrite them using two's complement representation instead.

These directives do not align the start of the memory allocated. If this is required you must use one of the alignment directives.

The following aliases for these directives are also accepted:

Table 2-78: Aliases for the data definition directives

Directive	Aliases
`.byte`	`.1byte` `.dc.b`

`.hword`	`.2byte`
	`.dc`
	`.dc.w`
	`.short`
	`.value`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Directive	Aliases

`.word`	`.4byte`
	`.long`
	`.int`
	`.dc.l`
	`.dc.a` (AArch32 only)

`.quad`	`.8byte` `.xword` (AArch64 only) `.dc.a` (AArch64 only)

Example: 8-bit memory allocation

```
// 8-bit memory location, initialized to 42:
    .byte 42
```

Example: 16-bit memory allocation

```
// 16-bit memory location, initialized to the difference between the
// 'start' and 'end' labels. They must both be defined in this assembly
// file, and must be in the same section as each other, but not
// necessarily the same section as this directive:
    .hword end - start
```

Example: 32-bit memory allocation

```
// 32-bit memory location, initialized to 15532:
    .word 15532

// 32-bit memory location, initailized to the address of an
// externally defined symbol:
    .word extern_symbol

// 32-bit memory location, containing the offset between the
// current location in the file and an externally defined symbol.
    .word extern_symbol - .
```

2.7.5 String definition directives

Allocates one or more bytes of memory in the current section, and defines the initial contents of the memory from a string literal.

Syntax

```
.ascii "<string>"

.asciz "<string>"

.string "<string>"
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<string>

The following escape characters are accepted in the string literal:

Table 2-79: Escape characters for the string definition directives

Escape character	Meaning
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\"	Quote (")
\\	Backslash (\)
\\<Octal_Escape_Code>	Three digit octal escape code for each ASCII character

Operation

The `.ascii` directive does not append a null byte to the end of the string.

The `.asciz` directive appends a null byte to the end of the string.

The `.string` directive is an alias for `.asciz`.

Example: Using a null-terminated string in a constant pool

```
  .text
hello:
  adr r0, str_hello
  b printf
str_hello:
  .asciz "Hello, world!\n"
```

This example generates pascal-style strings that are prefixed by a length byte and have no null terminator. The generated assembler uses a macro to avoid repeated code. See also Macro directives and Numeric local labels.

```
    .macro pascal_string, str
    .byte 2f - 1f
1:
    .ascii "\str"
2:
    .endm

    .data
hello:
    pascal_string "Hello"
goodbye:
    pascal_string "Goodbye"
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.7.6 Floating-point data definition directives

These directives allocate memory in the current section of the file, and define the initial contents of that memory using a floating-point value.

Syntax

```
.float <value> [, <value>]...

.double <value> [, <value>]...
```

Parameters

<value>

A floating-point value. If a floating-point value cannot be exactly represented by the storage format, it is rounded to the nearest representable value using the "round to nearest, ties to even" rounding mode.

Operation

The `.float` directive allocates 4 bytes of memory for each argument, and stores the values in IEEE754 single-precision format.

The `.double` directive allocates 8 bytes of memory for each argument, and stores the values in IEEE754 double-precision format.

The following aliases for these directives are also accepted:

Table 2-80: Aliases for the floating-point data definition directives

Directive	Alias
`.float`	`.single`, `.dc.s`
`.double`	`.dc.d`

Example: Float and double PI

```
float_pi:
  .float 3.14159265359
double_pi:
  .double 3.14159265359
```

2.7.7 Section directives

The section directives instruct the assembler to change the ELF section that code and data are emitted into.

Default

Some section names and section name prefixes implicitly have some flags set. Extra flags can be set using the flags argument, but it is not possible to clear these implicit flags. The section names that

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

have implicit flags are listed in the following table. For sections names not mentioned in the table, the default is to have no flags.

If the `%<type>` argument is not provided, the type is inferred from the section name. For sections names not mentioned in the following table, the default section type is `%progbits`.

Table 2-81: Sections with implicit flags and default types

Section name	Implicit Flags	Default Type
`.rodata`	a	`%progbits`
`.rodata.*`	a	`%progbits`
`.rodata1`	a	`%progbits`
`.text`	ax	`%progbits`
`.text.*`	ax	`%progbits`
`.init`	ax	`%progbits`
`.fini`	ax	`%progbits`
`.data`	aw	`%progbits`
`.data.*`	aw	`%progbits`
`.data1`	aw	`%progbits`
`.bss`	aw	`%nobits`
`.bss.*`	aw	`%nobits`
`.init_array`	aw	`%init_array`
`.init_array.*`	aw	`%init_array`
`.fini_array`	aw	`%fini_array`
`.fini_array.*`	aw	`%fini_array`
`.preinit_array`	aw	`%preinit_array`
`.preinit_array.*`	aw	`%preinit_array`
`.tdata`	awT	`%progbits`
`.tdata.*`	awT	`%progbits`
`.tbss`	awT	`%nobits`
`.tbss.*`	awT	`%nobits`
`.note*`	No default	`%note`

Syntax

```
.section <name> [, "<flags>" [, %<type> [, <entry_size>] [, <group_name> [,
 <linkage>]] [, <link_order_symbol>] [, <unique>, <unique_id>]]]

.pushsection .section <name> [, "<flags>" [, %<type> [, <entry_size>] [,
 <group_name> [, <linkage>]] [, <link_order_symbol>] [, <unique>, <unique_id>]]]

.popsection

.text

.data

.rodata

.bss
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<name>

The name of the section to switch to.

By default, if the name is identical to a previous section, or one of the built-in sections, the assembler switches back to that section. Any code or data that is assembled is appended to the end of that section. You can use the `<unique-id>` argument to override this behavior.

<flags>

A quoted string containing any of the following characters, which correspond to the sh_flags field in the ELF section header.

Table 2-82: Section flags

Flag	Meaning
a	SHF_ALLOC: the section is allocatable.
w	SHF_WRITE: the section is writable.
y	SHF_ARM_PURECODE: the section is not readable.
x	SHF_EXECINSTR: the section is executable.
o	SHF_LINK_ORDER: the section has a link-order restriction.
M	SHF_MERGE: the section can be merged.
S	SHF_STRINGS: the section contains null-terminated string.
T	SHF_TLS: the section is thread-local storage.
G	SHF_GROUP: the section is a member of a section group.
?	If the previous section was part of a group, this section is in the same group, otherwise it is ignored.

The flags can be specified as a numeric value, with the same encoding as the sh_flags in the ELF section header. This field cannot be combined with the flag characters listed in this table. When using this syntax, the quotes around the flags value are still required.


Note

Certain flags need extra arguments, as described in the respective arguments.

<type>

The optional `<type>` argument is accepted with two different syntaxes, `%<type>` and `"<type>"`. It corresponds to the sh_type field in the ELF section header. The following values for the type argument are accepted:

Table 2-83: Section Type

Argument	ELF type	Meaning
%progbits	SHT_PROGBITS	Section contains either initialized data and instructions or instructions only.
%nobits	SHT_NOBITS	Section contains only zero-initialized data.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Argument	ELF type	Meaning
`%note`	`SHT_NOTE`	Section contains information that the linker or loader use to check compatibility.
`%init_array`	`SHT_INIT_ARRAY`	Section contains an array of pointers to initialization functions.
`%fini_array`	`SHT_FINI_ARRAY`	Section contains an array of pointers to termination functions.
`%preinit_array`	`SHT_PREINIT_ARRAY`	Section contains an array of pointers to pre-initialization functions.

The type can be specified as a numeric value, with the same encoding as the sh_type in the ELF section header. When using this syntax, the quotes around the type value are still required.

`<entry_size>`

If the `M` flag is specified, the `<entry_size>` argument is required. This argument must be an integer value, which is the size of the records that are contained within this section, that the linker can merge.

`<group_name>`

If the `G` flag is specified, the `<group_name>` argument is required. This argument is a symbol name to be used as the signature to identify the section group. All sections in the same object file and with the same `<group_name>` are part of the same section group.

If the `?` flag is specified, the section is implicitly in the same group as the previous section, and the `<group_name>` and `<linkage>` options are not accepted.

It is an error to specify both the `G` and `?` flags on the same section.

`<linkage>`

If the `G` flag is specified, the optional linkage argument is allowed. The only valid value for this argument is `comdat`, which has the same effect as not providing the linkage argument. If any arguments after the `<group_name>` and `<linkage>` arguments are to be provided, then the linkage argument must be provided.

If the `?` flag is specified, the section is implicitly in the same group as the previous section, and the `<group_name>` and `<linkage>` options are not accepted.

It is an error to specify both the `G` and `?` flags on the same section.

`<link_order_symbol>`

If the `o` flag is specified, the `<link_order_symbol>` argument is required. This argument must be a symbol which is defined earlier in the same file. If multiple sections with the `o` flag are present at link time, the linker ensures that they are in the same order in the image as the sections that define the symbols they reference.

`<unique>` and `<unique_id>`

If the optional `<unique>` argument is provided, then the `<unique_id>` argument must also be provided. This argument must be a constant expression that evaluates to a positive integer. If a section has previously been created with the same name and unique ID, then the assembler

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

switches to the existing section, appending content to it. Otherwise, a new section is created. Sections without a unique ID specified are never merged with sections that do have one. Not merging allows creating multiple sections with the same name. The exact value of the unique ID is not important, and it has no effect on the generated object file.

Operation

The `.section` directive switches the current target section to the one described by its arguments. The `.pushsection` directive pushes the current target section onto a stack, and switches to the section described by its arguments. The `.popsection` directive takes no arguments, and reverts the current target section to the previous one on the stack. The remaining the directives, `.text`, `.data`, `.rodata`, and `.bss`, switch to one of the built-in sections.

If the `<name>` argument of the `.section` directive is the same as that of an existing section, but any of the `<flags>`, `<type>`, or `<entry_size>` arguments do not match the previous definition of the section, then an error is reported. If the `<flags>`, `<type>`, and `<entry_size>` arguments all match the previous definition of the section, then the existing section is extended, unless prevented by the `<unique>` and `<unique_id>` arguments.

Example: Splitting code and data into the built-in .text and .data sections

The linker can place the `.text` and `.data` sections independently, for example to place the code in flash memory, and the writable data in RAM:

```
      .text
get_value:
      movw r0, #:lower16:value
      movt r0, #:upper16:value
      ldr r0, [r0]
      bx lr

      .data
value:
      .word 42
```

Example: Create a section containing constant, mergeable records

This section contains a series of 8-byte records, where the linker is allowed to merge two records with identical content (possibly coming from different object files) into one record to reduce the image size.

```
      .section mergable, "aM", %progbits, 8
entry1:
      .word label1
      .word 42
entry2:
      .word label2
      .word 0x1234
```

Example: Creating two sections with the same name

```
      .section .data, "aw", %progbits, unique, 1
      .word 1
      .section .data, "aw", %progbits, unique, 2
      .word 2
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Creating a section group containing two sections

Here, the `G` flag is used for the first section, using the `group_signature` symbol. The second section uses the `?` flag to simplify making it part of the same group. Any other sections in this file using the `G` flag and `group_signature` symbol are placed in the same group.

```
      .section foo, "axG", %progbits, group_signature
get_value:
      movw r0, #:lower16:value
      movt r0, #:upper16:value
      ldr r0, [r0]
      bx lr

      .section bar, "aw?"
      .local value
value:
      .word 42
```

2.7.8 Conditional assembly directives

These directives allow you to conditionally assemble sequences of instructions and directives.

Syntax

```
.if[<modifier>] <expression>
    // ...
  [.elseif <expression>
    // ...]
  [.else
    // ...]
.endif
```

Parameters

`<modifier>`

The following forms of the `.if` directive are available, which check different conditions:

Table 2-84: .if condition modifiers

`.if` condition modifier	Meaning
`.if <expr>`	Assembles the following code if `<expr>` evaluates to non zero.
`.ifne <expr>`	Assembles the following code if `<expr>` evaluates to non zero.
`.ifeq <expr>`	Assembles the following code if `<expr>` evaluates to zero.
`.ifge <expr>`	Assembles the following code if `<expr>` evaluates to a value greater than or equal to zero.
`.ifle <expr>`	Assembles the following code if `<expr>` evaluates to a value less than or equal to zero.
`.ifgt <expr>`	Assembles the following code if `<expr>` evaluates to a value greater than zero.
`.iflt <expr>`	Assembles the following code if `<expr>` evaluates to a value less than zero.
`.ifb <text>`	Assembles the following code if the argument is blank.
`.ifnb <text>`	Assembles the following code if the argument is not blank.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

.if condition modifier	Meaning
.ifc <string1> <string2>	Assembles the following code if the two strings are the same. The strings can be optionally surrounded by double quote characters ("). If the strings are not quoted, the first string ends at the first comma character, and the second string ends at the end of the statement (newline or semicolon).
.ifnc <string1> <string2>	Assembles the following code if the two strings are not the same. The strings can be optionally surrounded by double quote characters ("). If the strings are not quoted, the first string ends at the first comma character, and the second string ends at the end of the statement (newline or semicolon).
.ifeqs <string1> <string2>	Assembles the following code if the two strings are the same. Both strings must be quoted.
.ifnes <string1> <string2>	Assembles the following code if the two strings are not the same. Both strings must be quoted.
.ifdef <expr>	Assembles the following code if symbol was defined earlier in this file.
.ifndef <expr>	Assembles the following code if symbol was not defined earlier in this file.

<expression>

An expression

Operation

There are many different forms of the `.if` directive that check different conditions. Each `.if` directive must have a matching `.endif` directive. A `.if` directive can optionally have one associated `.else` directive, and can optionally have any number of `.elseif` directives.

You can nest these directives, with the maximum nesting depth limited only by the amount of memory in your computer.

The `.elseif` directive takes an expression argument but does not take a condition modifier, and therefore always behaves the same way as `.if`, assembling the subsequent code if the expression is not zero, and if no previous conditions in the same `.if .elseif` chain were true.

The `.else` directive takes no argument, and the subsequent block of code is assembled if none of the conditions in the same `.if .elseif` chain were true.

Example: A macro to load an immediate value into a register

```
  // This macro expands to one or two instructions, depending
  // on the value of the immediate operand.
  .macro get_imm, reg, imm
    .if \imm >= 0x10000
      movw \reg, #\imm & 0xffff
      movt \reg, #\imm >> 16
    .else
      movw \reg, #\imm
    .endif
  .endm

  // The first of these macro invocations expands to one movw instruction,
  // the second expands to a movw and a movt instruction.
get_constants:
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
  get_imm r0, 42
  get_imm r1, 0x12345678
  bx lr
```

2.7.9 Macro directives

The `.macro` directive defines a new macro that you can use in other parts of your assembly. It can define a variable with a value or a sequence of instructions that you frequently use in your code.

Syntax

```
.macro <macro_name> [, <parameter_name>]...
  // ...
  [.exitm]
.endm
```

Parameters

`<macro_name>`

The name of the macro.

`<parameter_name>`

Inside the body of a macro, you can refer to the parameters by their name, prefixed with `\`. After instantiating the macro, parameter references expand to the value of the argument.

Parameters can be qualified in these ways:

Table 2-85: Macro parameter qualifier

Parameter qualifier	Meaning
`<name>=<value>`	Sets the default value for the parameter. If the argument in the instantiation is not provided or left blank, then the default value is used.
`<name>:req`	This qualifier marks the parameter as required, it is an error to instantiate the macro with a blank value for this parameter.
`<name>:vararg`	This parameter consumes all remaining arguments in the instantiation. If used, it must be the last parameter.

Operation

The `.macro` directive defines a new macro with name `macro_name`, and zero or more named parameters. The body of the macro extends to the matching `.endm` directive.

You can instantiate a defined macro by using it like an instruction mnemonic:

```
<macro_name> <argument>[, <argument>]...
```

Inside a macro body, `\@` expands to a counter value which is unique to each macro instantiation. You can use this value to create unique label names that do not interfere with other instantiations of the same macro.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The `.exitm` directive allows exiting a macro instantiation before reaching the end.

Example: Define a global integer macro

```
    // Macro for defining global variables, with the symbol binding, type, and
    // size set appropriately. You can omit the 'value' parameter, in which
    // case the variable gets an initial value of 0. It is an error to not
    // provide the 'name' argument.
    .macro global_int, name:req, value=0
    .global \name
    .type \name, %object
    .size \name, 4
\name:
    .word \value
    .endm

    .data
    global_int foo
    global_int bar, 42
```

2.7.10 Symbol binding directives

These directives modify the ELF binding of one or more symbols.

Syntax

```
.global <symbol>[, <symbol>]...

.local <symbol>[, <symbol>]...

.weak <symbol>[, <symbol>]...
```

Parameters

`<symbol>`

The ELF symbol to modify.

Operation

The `.global` directive sets the symbol binding to STB_GLOBAL. These symbols are visible to all object files being linked, so a definition in one object file can satisfy a reference in another.

The `.local` directive sets the symbol binding in the symbol table to STB_LOCAL. These symbols are not visible outside the object file they are defined or referenced in, so multiple object files can use the same symbol names without interfering with each other.

The `.weak` directive sets the symbol binding to STB_WEAK. These symbols behave similarly to global symbols, with these differences:

- It is not an error if a reference to a symbol with weak binding is not satisfied. That is, no definition of the symbol is found.
- It is not an error if multiple definitions of a weak symbol are present. If a definition of the symbol with strong binding is present, that definition satisfies all references to the symbol, otherwise one of the weak references is chosen.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The symbol binding directive can be at any point in the assembly file, before or after any references or definitions of the symbol.

If the binding of a symbol is not specified using one of these directives, the default binding is:

- If a symbol is not defined in the assembly file, it has global visibility by default.
- If a symbol is defined in the assembly file, it has local visibility by default.


Note

`.local` and `.L` are different directives. Symbols starting with `.L` are not put into the symbol table.

Example: A function with global binding, so that is can be referenced from other object files

```
    // The symbol 'value' defaults to local binding, so other object
    // files can use the symbol name 'value' without interfering with this
    // definition and reference.
    .global get_val
get_val:
    ldr r0, value
    bx lr
value:
    .word 0x12345678

    // The symbol 'printf' is not defined in this file, so defaults to global
    // binding, so the linker searches other object files and libraries to
    // find a definition of it.
    bl printf

    // The debug_trace symbol is a weak reference. If a definition of it is
    // found by the linker, this call is relocated to point to it. If a
    // definition is not found (for example in a release build that does
    // not include the debug code), the linker points the bl instruction at
    // the next instruction, so it has no effect.
    .weak debug_trace
    bl debug_trace
```

2.7.11 Org directive

The `.org` directive advances the location counter in the current section to new-location.

Syntax

```
.org <new_location> [, <fill_value>]
```

Parameters

`<new_location>`

Must be one of:

- An absolute integer expression, in which case it is treated as the number of bytes from the start of the section.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- An expression which evaluates to a location in the current section. This could use a symbol in the current section, or the current location ('.').

<fill_value>
An optional 1-byte value.

Operation

The `.org` directive can only move the location counter forward, not backward.

By default, the `.org` directive inserts zero bytes in any locations that it skips over. This can be overridden using the optional `<fill_value>` argument, which sets the 1-byte value that is repeated in each skipped location.

Example: Macro to create one AArch64 exception vector table entry

```
    // Each entry must be 128 bytes in length. If the code is shorter
    // than that, padding is inserted. If the code is longer than that,
    // the .org directive reports an error, as this would require the
    // location counter to move backwards.
    .macro exc_tab_entry, num
1:
    mov x0, #\num
    b unhandled_exception
    .org 1b + 0x80
    .endm

    // Each of these macro instantiations emits 128 bytes of code and padding.
    .section vectors, "ax"
    exc_tab_entry 0
    exc_tab_entry 1
    // More table entries...
```

2.7.12 AArch32 target selection directives

The AArch32 target selection directives specify code generation parameters for AArch32 targets. You can change between the A32 and T32 instruction sets and specify build attributes for use by `armlink`.

Syntax

```
.arm

.thumb

.arch <arch_name>

.cpu <cpu_name>

.fpu <fpu_name>

.arch_extension <extension_name>

.eabi_attribute <tag>, <value>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<arch_name>

The same names as the `-march` command-line option, but does not accept the optional architecture extensions. Use `-march=list` to see a list of the architecture names.

<cpu_name>

The same names as the `-mcpu` command-line option, but does not accept the optional architecture extensions. Use `-mcpu=list` to see a list of the processor names.

<fpu_name>

The same names as the `-mfpu` command-line option. For AArch32 targets, use `-mfpu=list` to see a list of the FPU names.

<extension_name>

The optional extension to the architecture or processor. It accepts one of the following optional extensions:

- `crc`
- `fp16`
- `ras`

To disable an extension, prefix the `<extension_name>` with `no`.

<tag>

The `<tag>` argument specifies the tag that is to be set. This argument can be either the tag name or tag number, but not both.

<value>

The `<value>` argument specifies the value to set for the `<tag>`. The value can be either an integer or string type. The type must match the type expected for that tag.

Operation

.arm

The `.arm` directive instructs the assembler to interpret subsequent instructions as A32 instructions, using the UAL syntax.

The `.code 32` directive is an alias for `.arm`.

.thumb

The `.thumb` directive instructs the assembler to interpret subsequent instructions as T32 instructions, using the UAL syntax.

The `.code 16` directive is an alias for `.thumb`.

.arch

The `.arch` directive changes the architecture for which the assembler generates instructions.

.cpu

The `.cpu` directive changes the CPU for which the assembler generates instructions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`.fpu`

The `.fpu` directive changes the FPU for which the assembler generates instructions.

`.arch_extension`

The `.arch_extension` directive enables or disables the optional extensions to the architecture or CPU for which the assembler generates instructions.

`.eabi_attribute`

The `.eabi_attribute` directive sets a build attribute in the output file. Build attributes are used by `armlink` to check for co-compatibility between object files, and to select suitable libraries.

The `.eabi_attribute` directive does not affect which instructions the assembler accepts. We recommend that the `.arch`, `.cpu`, `.fpu`, and `.arch_extension` directives are used where possible. These directives are recommended because they also make sure that instructions for the selected architecture are valid. These directives also set the relevant build attributes, so the `.eabi_attribute` directive is only needed for attributes not covered by them.


Note

`Tag_compatibility` is a special tag that requires both an integer value and a string value:

```
      .eabi_attribute Tag_compatibility, <integer_value>,
<string_value>
```

Example: Generate code for the Armv7-M architecture

```
      .arch armv7-m
```

Example: Generate code for the Cortex-R5, without an FPU

```
      .cpu cortex-r5
      .fpu none
```

Example: Generate code for Armv8.2-A with the FP16 extension

```
      .arch armv8.2-a
      .fpu neon-fp-armv8
      .arch_extension fp16
```

Example: Generate code for Armv8-A with the RAS extension

```
      .arch armv8-a
      .arch_extension ras
      esb
```

Related information

-march on page 123

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.7.13 AArch64 target selection directives

The AArch64 target selection directives specify code generation parameters for AArch64 targets.

Syntax

```
.arch <arch_name>[+[no]<extension>]...
.cpu <cpu_name>[+[no]<extension>]...
.arch_extension <extension_name>
```

Parameters

<arch_name>

The `<arch_name>` argument accepts the same values as the `-march` option. See -march for more information.

<cpu_name>

The `<cpu_name>` argument accepts the same values as the `-mcpu` option. See -mcpu for more information.

<extension>

One or more optional architecture extensions:

- For supported `.arch` extensions, see -march.
- For supported `.cpu` extensions, see -mcpu.

To disable an extension, prefix the `<extension>` with `no`.

<extension_name>

The optional extension to the architecture or processor. `.arch_extension` accepts one of the following optional extensions:

- `b16b16` (FEAT_SVE_B16B16) [ALPHA] support
- `cpa` (FEAT_CPA)
- `crc` (FEAT_CRC32)
- `crypto`
- `cssc` (FEAT_CSSC)
- `d128` (FEAT_D128)
- `fp` (FEAT_FP)
- `gcs` (FEAT_GCS) [ALPHA] support
- `ite` (FEAT_ITE)
- `ls64` (FEAT_LS64, FEAT_LS64_V, and FEAT_LS64_ACCDATA)
- `lse128` (FEAT_LSE128)
- `pauth` (FEAT_PAuth)
- `pauth-lr` (FEAT_PAuth_LR)

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- `predres2` (FEAT_SPECRES2)
- `ras` (FEAT_RAS and FEAT_RASv1p1)
- `rasv2` (FEAT_RASv2)
- `rcpc3` (FEAT_LRCPC3)
- `sve2p1` (FEAT_SVE2p1)
- `sme2p1` (FEAT_SME2p1)
- `sme-f64f64` (FEAT_SME_F6464)
- `sme-i16i64` (FEAT_SME_I16I64)
- `sme-f16f16` (FEAT_SME_F16F16)
- `simd` (FEAT_AdvSIMD)
- `the` (FEAT_THE)
- `tlbiw` (FEAT_TLBIW)

To disable an extension, prefix the `<extension_name>` with `no`.

Restrictions

`.arch` and `.cpu` directives that do not explicitly include or exclude an extension might have no effect.

For example:

```
// test.s
    .arch armv8-a
    esb    // invalid without the RAS extension
```

Assemble `test.s` using the following command:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.2-a -c test.s -o test.o
```

The `-march=armv8.2-a` option specifies that the target architecture is Arm®v8.2-A. Armv8.2-A includes the *Reliability, Availability, and Serviceability* (RAS) extension by default. However, the `.arch` directive in `test.s` specifies the Armv8-A architecture, which does not support RAS by default. Therefore, you might expect the `.arch armv8-a` directive to override the `-march=armv8.2-a` option and disable the RAS extension. The compiler ignores `.arch armv8-a` and therefore does not disable the RAS extension. As a result, the compiler reports an error, for example:

```
test.s:2:7: error: instruction requires: ras
      esb    // invalid without the RAS extension
      ^
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

To avoid this issue, you must always explicitly include or exclude an extension with `.arch` or `.cpu` directives. For example, change the `.arch` directive in `test.s` to:

```
// test.s
    .arch armv8-a+ras
    esb
```

The compiler does not now report an error.

Operation

`.arch`

The `.arch` directive changes the architecture for which the assembler generates instructions.

`.cpu`

The `.cpu` directive changes the CPU for which the assembler generates instructions.

`.arch_extension`

The `.arch_extension` directive enables or disables the optional extensions to the architecture or processor for which the assembler generates instructions.

Example: Generate code for Armv8-A without a floating-point unit

```
// The assembler reports an error if any instructions
// following this directive require the floating-point unit.
    .arch armv8-a+nofp
```

Related information

-march on page 123
Feature names in A-profile architecture

2.7.14 Space-filling directives

The `.space` directive emits `<count>` bytes of data, each of which has value `<value>`. If the `<value>` argument is omitted, it defaults to zero.

Syntax

```
.space <count> [, <value>]

.fill <count> [, <size> [, <value>]]
```

Parameters

`.space` parameters

The `.space` directive emits `<count>` bytes of data, each of which has value `<value>`. If the `<value>` argument is omitted, its default value is zero.

The `.skip` and `.zero` directives are aliases for the `.space` directive.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`.fill` parameters

The `.fill` directive emits `<count>` data values, each with length `<size>` bytes and value `<value>`. If `<size>` is greater than 8, it is truncated to 8. If the `<size>` argument is omitted, its default value is one. If the `<value>` argument is omitted, its defaults value is zero.

The `.fill` directive always interprets the `<value>` argument as a 32-bit value:

- If the `<size>` argument is less than or equal to 4, the `<value>` argument is truncated to `<size>` bytes, and emitted with the appropriate endianness for the target. The assembler does not emit a diagnostic if `<value>` is truncated in this case.
- If the `<size>` argument is greater than 4, the value is emitted as a 4-byte value with the appropriate endianness. The value is emitted in the 4 bytes of the allocated memory with the lowest addresses. The remaining bytes in the allocated memory are then filled with zeroes. In this case, the assembler does emit a diagnostic if the value is truncated.

2.7.15 Type directive

The `.type` directive sets the type of a symbol.

Default

The default type is `%object`.

Syntax

```
.type <symbol>, %<type>
```

Parameters

`<symbol>`

The symbol name for which to set the type.

`%<type>`

One of the following:

- `%function`
- `%object`
- `%tls_object`

Example: Function type symbol

```
  // 'func' is a function
  .type func, %function
func:
  bx lr
```

Example: Object type symbol

```
  // 'value' is a data object:
  .type value, %object
value:
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    .word 42
```

2.7.16 Call Frame Information directives

Call Frame Information (CFI) directives are required for debugging. Most assembler functions do not use the stack. Therefore, a backtrace is all that is required.

To get a backtrace for your GNU-syntax assembly code, you need only use the following subset of `.cfi` directives:

Table 2-86: CFI directives

CFI directives	Meaning
`.cfi_sections .debug_frame`	Emits the `.debug_frame` section. This directive tells the assembler to write out a section of debug frame data.
`.cfi_startproc`	Include this directive at the start of each function that is to have an entry in `.debug_frame`. Make sure you close each of those functions with `.cfi_endproc`.
`.cfi_endproc`	Include this directive at the end of each function that also includes `.cfi_startproc`.

The `.cfi_startproc` and `.cfi_endproc` directives indicate that the frame data must consider the range in between them to be an individual function.

Related information

How to get a backtrace through assembler functions
CFI directives

2.7.17 A32 and T32 instruction substitutions

In certain circumstances, if the value of an Operand2 constant is not available with a given instruction, but its logical inverse or negation is available, then `armclang` can produce an equivalent instruction with the inverted or negated constant.

This substitution applies to both assembly language source files and to inline assembly code in C and C++ language source files.

More information about the syntax of Operand2 constants is available in the *Arm Instruction Set for Armv7 and earlier Arm architectures Reference Guide*. The following table shows the instruction substitutions supported by `armclang`, based on the values of Operand2 constants for the A32 and T32 instruction sets.

Table 2-87: A32 and T32 instruction substitutions

A32 and T32 instruction	Equivalent instruction	Constant substitution method
`ADC`*`Scond Rd`*`, <Rn>, #<constant>`	`SBC`*`Scond Rd`*`, <Rn>, #~<constant>`	Logical inversion
`ADD`*`Scond Rd`*`, <Rn>, #<constant>`	`SUB`*`Scond Rd`*`, <Rn>, #-<constant>`	Negation

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A32 and T32 instruction	Equivalent instruction	Constant substitution method
`AND`*`cond`* `<Rd>, <Rn>, #<constant>`	`BIC`*`cond`* `<Rd>, <Rn>, #~<constant>`	Logical inversion
`BIC`*`cond`* `<Rd>, <Rn>, #<constant>`	`AND`*`cond`* `<Rd>, <Rn>, #~<constant>`	Logical inversion
`CMP`*`cond`* `<Rn>, #<constant>`	`CMN`*`cond`* `<Rn>, #-<constant>`	Negation
`CMN`*`cond`* `<Rn>, #<constant>`	`CMP`*`cond`* `<Rn>, #-<constant>`	Negation
`MOV`*`cond`* `<Rd>, #<constant>`	`MVN`*`cond`* `<Rd>, #~<constant>`	Logical inversion
`MVN`*`cond`* `<Rd>, #<constant>`	`MOV`*`cond`* `<Rd>, #~<constant>`	Logical inversion
`ORN`*`cond`* `<Rd>, <Rn>, #<constant>` (T32 only)	`ORR`*`cond`* `<Rd>, <Rn>, #~<constant>` (T32 only)	Logical inversion
`ORR`*`cond`* `<Rd>, <Rn>, #<constant>` (T32 only)	`ORN`*`cond`* `<Rd>, <Rn>, #~<constant>` (T32 only)	Logical inversion
`SBC`*`cond`* *`Rd`*`, <Rn>, #<constant>`	`ADC`*`cond`* *`Rd`*`, <Rn>, #~<constant>`	Logical inversion
`SUB`*`cond`* *`Rd`*`, <Rn>, #<constant>`	`ADD`*`cond`* *`Rd`*`, <Rn>, #-<constant>`	Negation

You can disable this substitution using the `armclang` option `-mno-neg-immediates`.

Related information

-mno-neg-immediates on page 184

2.7.18 Integrated assembler support for the CSDB instruction

For conditional CSDB instructions that specify a condition `<c>` other than `AL` in A32, and for any condition `{c}` used inside an `IT` block in T32, `armclang` rejects conditional CSDB instructions, outputs an error message, and aborts.

For example:

```
<stdin>:10:7: error: instruction 'csdb' is not predicable, but condition code
 specified
  csdbeq
      ^
```

The same error is output for both A32 and T32.

Related information

CSDB instruction

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.8 armclang inline assembler

The inline assembler provides a mechanism for manually inserting assembly instructions into C and C++ code without having to write the entire function in assembly code. Inline assembly in Arm® Compiler for Embedded 6 must be written in GNU assembly syntax.

Width specifiers in inline assembly

There are some issues when using width specifiers in inline assembly instructions:

T32 instruction with an invalid `.n` width specifier

The inline assembler incorrectly fails to report an error for a T32 instruction with an invalid `.n` width specifier. Instead, the inline assembler assembles the instruction as a 32-bit instruction.

For example, the inline assembler incorrectly fails to report an error for the following instruction:

```
    adc.n r0, r1, #1
```

Instruction includes the `.w` width specifier

The inline assembler might report an error if an instruction includes the `.w` width specifier.

For example, create the C program `adc.c` containing the `ADC` instruction `adc.w r0, r1, 0x99`:

```
int main(void) {
    __asm("adc.w r0, r1, 0x99");
    return 0;
}
```

Compiling the program gives an error:

```
armclang --target=arm-arm-none-eabi -march=armv7-m -S -o adc.s adc.c
adc.c:2:11: error: invalid instruction, any one of the following would fix
 this:
    __asm("adc.w r0, r1, 0x99");
          ^
<inline asm>:1:2: note: instantiated into assembly here
        adc.w r0, r1, 0x99
        ^
adc.c:2:11: note: invalid operand for instruction
    __asm("adc.w r0, r1, 0x99");
          ^
...
1 error generated.
```

2.8.1 Inline Assembly

`armclang` provides an inline assembler that enables you to write assembly language sequences in C and C++ language source files. The inline assembler also provides access to features of the target processor that are not available from C or C++.

You can use inline assembly in two contexts:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- File-scope inline assembly statements.

```
__asm(".global __use_realtime_heap");
```

- Inline assembly statement within a function.

```
void set_translation_table(void *table) {
  __asm("msr TTBR0_EL1, %0"
    :
    : "r" (table)
  );
}
```

Both syntaxes accept assembly code as a string. Write your assembly code in the syntax that the integrated assembler accepts. For both syntaxes, the compiler inserts the contents of the string into the assembly code that it generates. All assembly directives that the integrated assembler accepts are available to use in inline assembly. However, the state of the assembler is not reset after each block of inline assembly. Therefore, avoid using directives in a way that affects the rest of the assembly file, for example by switching the instruction set between A32 and T32. See armclang Integrated Assembler.

Implications for inline assembly with optimizations

You can write complex inline assembly that appears to work at some optimization levels, but the assembly is not correct. The following examples describe some situations when inline assembly is not guaranteed to work:

- Including an instruction that generates a literal pool. There is no guarantee that the compiler can place the literal pool in the range of the instruction.
- Using or referencing a function only from the inline assembly without telling the compiler that it is used. The compiler treats the assembly as text. Therefore, the compiler can remove the function that results in an unresolved reference during linking. The removal of the function is especially visible for *Link-Time Optimization* (LTO), because LTO performs whole program optimization and is able to remove more functions.

  For file-scope inline assembly, you can use the `__attribute((used))` function attribute to tell the compiler that a function is used. For inline assembly statements, use the input and output operands.
- Functions containing inline assembly which do not access any volatile variables. Use the `volatile` keyword to ensure the code block containing your inline assembly is not removed.

For large blocks of assembly code where the overhead of calling between C or C++ and assembly is not significant, we recommend using a separate assembly file, which does not have these limitations.

Inline assembly does not define higher bits when short or char is used

In general, with inline assembly, there is no requirement for the operand width to match the register width.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Using inline assembly code means that the compiler puts the value of the expression in the allocated register without converting that value. Therefore, the spare register bits have unspecified values.

Example: LDRH instruction loads an unsigned halfword value and zero-extends the remaining high bits

In the following example, the `LDRH` instruction in the assembly output loads the unsigned halfword value in `s->x` into r0, and zero-extends the remaining high bits.

1. Create the `func.c` file containing the following C code:

```
struct S {
    short x, y;
};

void f(struct S *s, int *p) {
    int r;
    __asm__ __volatile__ ("mov %0, %1"
                          : "=r" (r)
                          : "r" (s->x));
    *p = r;
}

void g(struct S s, int *p) {
    f(&s, p);
}
```

2. Compile with:

```
armclang --target=arm-arm-none-eabi -Ofast -mcpu=cortex-m33 -S func.c
```

   The resulting assembly file `func.s` contains:

```
...
f:
...
    ldrh    r0, [r0]
    @APP
    mov     r0, r0
    @NO_APP
    str     r0, [r1]
    bx      lr
...
g:
...
    @APP
    mov     r0, r0
    @NO_APP
    str     r0, [r1]
    bx      lr
...
```

Related information

File-scope inline assembly on page 404
Output and input operands on page 407
Optimizing across modules with Link-Time Optimization

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.8.2 File-scope inline assembly

Inline assembly can be used at file-scope to insert assembly into the output of the compiler.

All file-scope inline assembly code is inserted into the output of the compiler before the code for any functions or variables declared in the file, regardless of where they appear in the input. If multiple blocks of file-scope inline assembly code are present in one file, they are emitted in the same order as they appear in the source code.

Compiling multiple files containing file-scope inline assembly with the `-flto` option does not affect the ordering of the blocks within each file, but the ordering of blocks in different files is not defined.

Syntax

```
__asm("<assembly code>");
```

If you include multiple assembly statements in one file-scope inline assembly block, you must separate them by newlines or semicolons. The assembly string does not have to end in a new-line or semicolon.

Restrictions

When using assembly instructions in file-scope inline assembly and *Link-Time Optimization* (LTO), the compiler does not always report an error for instructions that are not available for the selected target features. Also, the compiler might report an error for instructions that are available for the selected target features.

Example: Simple file-scope inline assembly

```
// Simple file-scope inline assembly.
__asm(".global __use_realtime_heap");
```

Example: Multiple file-scope inline assembly statements in one block

```
// Multiple file-scope inline assembly statements in one block:
__asm("add_ints:\n"
    "  add r0, r0, r1\n"
    "  bx lr");
```

Example: C++11 raw string literals

```
// C++11 raw string literals can be used for long blocks,
// without needing to include escaped newlines in the
// assembly string (requires C++11):
__asm(R"(
  sub_ints:
    sub r0, r0, r1
    bx lr
)");
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Error not reported for file-scope inline assembly and LTO when instruction is invalid for the target

Create the file `test.c` containing the code:

```
asm("vmov s0, s1");
```

Compile the program without LTO and for the Arm®v7-A architecture. This architecture enables *Floating-Point* (FP) by default, so disable it with `+nofp`:

```
armclang --target=arm-arm-none-eabi -march=armv7-a+nofp -mfpu=fpv5-d16 -c test.c
<inline asm>:1:1: error: instruction requires: VFP2
    1 | vmov s0, s1
      | ^
1 error generated.
```

Compile the program with LTO:

```
armclang --target=arm-arm-none-eabi -march=armv7-a+nofp -mfpu=fpv5-d16 -c test.c -
flto
```

No error is generated in this case even though the `vmov` instruction is invalid for the specified target.

Example: Error reported for file-scope inline assembly and LTO when instruction is valid for the target

Compile the program without LTO for the Armv7-M architecture and FPv5-D16 Floating-point extension:

```
armclang --target=arm-arm-none-eabi -march=armv7-m -mfpu=fpv5-d16 -c test.c
```

No error is generated.

Compile the program with LTO:

```
armclang --target=arm-arm-none-eabi -march=armv7-m -mfpu=fpv5-d16 -c test.c -flto
<inline asm>:1:1: error: instruction requires: VFP2
    1 | vmov s0, s1
      | ^
1 error generated.
```

The `vmov` instruction is rejected even though the `vmov` instruction is valid for the specified target.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.8.3 Inline assembly statements within a function

Inline assembly statements can be used inside a function to insert assembly code into the body of a C or C++ function.

Inline assembly code allows for passing control-flow and values between C/C++ and assembly at a fine-grained level. The values that are used as inputs to and outputs from the assembly code must be listed. Special tokens in the assembly string are replaced with the registers that contain these values.

As with file-scope inline assembly, you can use any instructions that are available in the integrated assembler in the assembly string. You can use most directives, however the `.section` directive is not allowed, and generates an error. You can use `.section` at file-scope. Use multiple assembly statements in the string of one inline assembly statement by separating them with newlines or semicolons. If you use multiple instructions in this way, the optimizer treats them as a complete unit. It does not split them up, reorder them, or omit some of them.

The compiler does not guarantee that the ordering of multiple inline assembly statements is preserved. It might also do the following:

- Merge two identical inline assembly statements into one inline assembly statement.
- Split one inline assembly statement into two inline assembly statements.
- Remove an inline assembly statement that has no apparent effect on the result of the program.

To prevent the compiler from doing any of these operations, you must use the input and output operands and the `volatile` keyword to indicate to the compiler which optimizations are valid.

The compiler does not parse the contents of the assembly string, except for replacing template strings, until code generation is complete. The compiler relies on the input and output operands, and clobbers to tell it about the requirements of the assembly code, and constraints on the surrounding generated code. Therefore the input and output operands, and clobbers must be accurate.

Syntax

```
__asm [volatile] (
  "<assembly string>"
   [ : <output operands>
   [ : <input operands>
   [ : <clobbers> ] ] ]
  );
```

2.8.3.1 Assembly string

An assembly string is a string literal that contains the assembly code.

The assembly string can contain template strings, starting with `%`, which the compiler replaces. The main use of these strings is to use registers that the compiler allocates to hold the input and output operands.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

Template strings for operands can take one of the following forms:

```
"%<modifier><number>"
"%<modifier>[<name>]"
```

`<modifier>` is an optional code that modifies the format of the operand in the final assembly string. You can use the same operand multiple times with different modifiers in one assembly string. See Inline assembly template modifiers.

For numbered template strings, the operands of the inline assembly statement are numbered, starting from zero, in the order they appear in the operand lists. Output operands appear before input operands.

If an operand has a name in the operand lists, you can use this name in the template string instead of the operand number. Square brackets must surround the name. Using names makes larger blocks of inline assembly easier to read and modify.

The `%%` template string emits a `%` character into the final assembly string.

The `%=` template string emits a number that is unique to the instance of the inline assembly statement. See Duplication of labels in inline assembly statements.

2.8.3.2 Output and input operands

The inline assembly statement can optionally accept two lists of operand specifiers, the first for outputs and the second for inputs. These lists are used to pass values between the assembly code and the enclosing C/C++ function.

Syntax

Each list is a comma-separated list of operand specifiers. Each operand specifier can take one of the following two forms:

```
[<name>] "<constraint>" (<value>)
         "<constraint>" (<value>)
```

Parameters

`<name>`

Is a name for referring to the operand in templates inside the inline assembly string. If the name for an operand is omitted, it must be referred to by number instead.

`<constraint>`

Is a string that tells the compiler how the value is used in the assembly string, including:

- For output operands, whether it is only written to, or both read from and written to. Also whether it can be allocated to the same register as an input operand. See Constraint modifiers.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Whether to store the value in a register or memory, or whether it is a compile-time constant. See Constraint codes.

<value>

Is a C/C++ value that the operand corresponds to. For output operands, this value must be a writable value.

Example: Saturating add output and input operands

Create the file `saturating_add.c` containing:

```
//saturating_add.c

int saturating_add(int a, int b) {
  int result;
  __asm(
      // The assembly string uses templates for the registers which hold output
      // and input values. These are replaced with the names of the
      // registers that the compiler chooses to hold the output and input
      // values.

      "qadd %0, %[lhs], %[rhs]"

      // The output operand, which corresponds to the "result" variable. This
      // does not have a name assigned, so must be referred to in the assembly
      // string by its number ("%0").
      // The "=" character in the constraint string tells the compiler that the
      // register chosen to hold the result does not need to have any
      // particular value at the start of the inline assembly.
      // The "r" character in the constraint tells the compiler that the value
      // must be placed in a general-purpose register (r0-r12 or r14).

      : "=r" (result)

      // The two input operands also use the "r" character in their
      // constraints, so the compiler places them in general-purpose
      // registers.
      // These have names specified, which can be used to refer to them in
      // the assembly string ("%[lhs]" and "%[rhs]").

      : [lhs] "r" (a), [rhs] "r" (b)
  );

  return result;
}
```

Build this example with the following command:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -c -S saturating_add.c -o
 saturating_add.s
```

The assembly language source file `saturating_add.s` that is generated contains:

```
    .section    .text.saturating_add,"ax",%progbits
    .hidden saturating_add
    .globl  saturating_add
    .p2align    2
    .type       saturating_add,%function
    .code  32
saturating_add:
   ...
    @APP
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    qadd  r0, r0, r1
    @NO_APP
    bx    lr
   ...
```

In this example:

- The compiler places the C function `saturating_add()` in a section that is called `.text.saturating_add`.
- Within the body of the function, the compiler expands the inline assembly statement into the `qadd r0, r0, r1` instruction between the comments `@APP` and `@NO_APP`. In `-S` output, the compiler always places code that it expands from inline assembly statements within a function between a pair of `@APP` and `@NO_APP` comments.
- The compiler uses the general-purpose register R0 for:
  - The `int a` parameter of the `saturating_add()` function.
  - The inline assembly input operand `%[lhs]`.
  - The inline assembly output operand `%0`.
  - The return value of the `saturating_add()` function.
- The compiler uses the general-purpose register R1 for:
  - The `int b` parameter of the `saturating_add()` function.
  - The inline assembly input operand `%[rhs]`.

2.8.3.3 Clobber list

The clobber list is a comma-separated list of strings. Each string is the name of a register that the assembly code potentially modifies, but for which the final value is not important.

To prevent the compiler from using a register for a template string in an inline assembly string, add the register to the clobber list.

For example, if a register holds a temporary value, include it in the clobber list. The compiler avoids using a register in this list as an input or output operand, or using it to store another value when the assembly code is executed.

In addition to register names, you can use two special names in the clobber list:

`"memory"`
: This string tells the compiler that the assembly code might modify any memory, not just variables that are included in the output constraints.

`"cc"`
: This string tells the compiler that the assembly code might modify any of the condition flags N, Z, C, or V. In AArch64 state, these condition flags are in the NZCV register. In AArch32 state, these condition flags are in the `CPSR` register.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

```
void enable_aarch64() {
  // Set bit 10 of SCR_EL3, to enale AArch64 at EL2.
  __asm volatile(R"(
        mrs x0, SCR_EL3
        orr x0, x0, #(1<<10)
        msr SCR_EL3, x0
        )" : /* no outputs */ : /* no inputs */
        // We used x0 as a temporary register, so we need to mark it as
        // clobbered, to prevent the compiler from storing a value in it.
      : "x0");
}
```

2.8.3.4 volatile

The optional `volatile` keyword tells the compiler that the assembly code has side-effects that the output, input, and clobber lists do not represent. For example, use this keyword with inline assembly code that sets the value of a System register.

Example: Use of the volatile keyword with inline assembly

The following is an example where the `volatile` keyword is required. If the `volatile` keyword is omitted, this example appears to still work. However, if the compiler inlines the code into a function that does not use the return value (`old_table`), then the inline assembly statement appears to be unnecessary, and could get optimized out. The `volatile` keyword lets the compiler know that the assembly has an effect other than providing the output value, so that this optimization does not happen.

```
void *swap_ttbr0(void *new_table) {
  void *old_table;
  __asm volatile (
    "mrs %[old], TTBR0_EL1\n"
    "msr TTBR0_EL1, %[new]\n"
    : [old] "=&r" (old_table)
    : [new] "r"   (new_table));
  return old_table;
}
```

Example: What happens when volatile is not used for inline assembly

The following example shows what happens when `volatile` is not used:

- Create the file `test.c` containing the following code:

```
#include <stdint.h>

static uint32_t without_volatile(uint32_t CONTROL_new)
{
  uint32_t CONTROL_old;

  __asm(
    "mrs %0, CONTROL\n"
    "msr CONTROL, %1\n"
    "isb\n"
    : "=&r" (CONTROL_old)
    : "r" (CONTROL_new)
  );
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
  return CONTROL_old;
}

static uint32_t with_volatile(uint32_t CONTROL_new)
{
  uint32_t CONTROL_old;

  __asm volatile(
    "mrs %0, CONTROL\n"
    "msr CONTROL, %1\n"
    "isb\n"
    : "=&r" (CONTROL_old)
    : "r" (CONTROL_new)
  );

  return CONTROL_old;
}

void test(void)
{
  /* When without_volatile is inlined,
     the __asm statement is removed.
   */
  without_volatile(0);

  /* When write_with_volatile is inlined,
     the __asm statement is preserved.
   */
  with_volatile(1);
}
```

- Compile `test.c` with:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m7 -O2 -std=c90 -c test.c -o
 test.o
```

  The compiler inlines the calls to the functions `without_volatile()` and `with_volatile()` into `test()`. However, because `test()` does not use the return value of `without_volatile()`, the compiler removes the call to `without_volatile()`.

- Use the `fromelf` utility to see the result:

```
fromelf --text -c test.o

...
** Section #3 '.text.test' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 16 bytes (alignment 8)
    Address: 0x00000000

    $t.0
    [Anonymous symbol #3]
    test
        0x00000000:    2001        .       MOVS     r0,#1
        0x00000002:    f3ef8114    ....    MRS      r1,CONTROL
        0x00000006:    f3808814    ....    MSR      CONTROL,r0
        0x0000000a:    f3bf8f6f    ..o.    ISB
        0x0000000e:    4770        pG      BX       lr
...
```

For this example, the CONTROL register is never set to the value 0, and could result in unexpected runtime behavior.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.8.4 Inline assembly constraint strings

A constraint string is a string literal, the contents of which are composed of two parts.

The contents of the constraint string are:

- A constraint modifier if the constraint string is for an output operand.
- One or more constraint codes.

2.8.4.1 Constraint modifiers

All output operands require a constraint modifier. There are currently no supported constraint modifiers for input operands.

Table 2-88: Constraint modifiers

Modifier	Meaning
=	This operand is only written to, and only after all input operands have been read for the last time. Therefore, the compiler can allocate this operand and an input to the same register or memory location.
+	This operand is both read from and written to.
=&	This operand is only written to. It might be modified before the assembly block finishes reading the input operands. Therefore, the compiler cannot use the same register to store this operand and an input operand. Operands with the =& constraint modifier are known as early-clobber operands. **Note:** In the case where a register constraint operand and a memory constraint operand are used together, you must use the =& constraint modifier on the register constraint operand to prevent the register from being used in the code generated to access the memory.
+&	This operand is written to, and might also be read from. Operands with the +& constraint modifier are known as early-clobber operands. An early-clobber operand is one that is written to before all input operands are read. If the early-clobber operand is also a read/write operand, the operand is only written to after it is read.

2.8.4.2 Constraint codes

Constraint codes define how to pass an operand between assembly code and the surrounding C or C++ code.

There are three categories of constraint codes:

Constant operands

You can only use these operands as input operands, and they must be compile-time constants. Use where a value is used as an immediate operand to an instruction. There are target-specific constraints that accept the immediate ranges suitable for different instructions.

Register operands

You can use these operands as both input and output operands. The compiler allocates a register to store the value. As there are a limited number of registers, it is possible to write an

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

inline assembly statement for which there are not enough available registers. In this case, the compiler reports an error. The exact number of available registers varies depending on the target architecture and the optimization level.

Memory operands

You can use these operands as both input and output operands. Use them with load and store instructions. Usually a register is allocated to hold a pointer to the operand. As there are a limited number of registers, it is possible to write an inline assembly statement for which there are not enough available registers. In this case, the compiler reports an error. The exact number of available registers can vary depending on the target architecture and the optimization level.

There are some common constraints, which can be used in both AArch32 state and AArch64 state. Other constraints are specific to AArch32 state or AArch64 state. In AArch32 state, there are some constraints that vary depending on the selected instruction set.

2.8.4.3 Constraint codes common to AArch32 state and AArch64 state

The following constraint codes are common to both AArch32 state and AArch64 state.

Constants

`i`

A constant integer, or the address of a global variable or function.

`n`

A constant integer (non-relocatable values only).

`s`

A constant integer (relocatable values only).


Note

The immediate constraints only check that their operand is constant after optimizations have been applied. Therefore it is possible to write code that you can only compile at higher optimization levels. We recommend that you test your code at multiple optimization levels to ensure it compiles.

Memory

`m`

A memory reference. This constraint causes a general-purpose register to be allocated to hold the address of the value instead of the value itself. By default, this register is printed as the name of the register surrounded by square brackets, suitable for use as a memory operand. For example, `[r4]` or `[x7]`. In AArch32 state only, you can print the register without the surrounding square brackets by using the `m` template modifier. See Template modifiers for AArch32 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Other

x

If the operand is a constant after optimizations have been performed, this constraint is equivalent to the `i` constraint. Otherwise, it is equivalent to the `r` or `w` constraints, depending on the type of the operand.


Note

We recommend that you use more precise constraints where possible. The `x` constraint does not perform any of the range checking or register restrictions that the other constraints perform.

2.8.4.4 Constraint codes for AArch32 state

The following constraint codes are specific to AArch32 state.

Registers

r

Operand must be an integer or floating-point type.

For targets that do not support Thumb®-2 technology, the compiler can use R0-R7.

For all other targets, the compiler can use R0-R12, or R14.

l

Operand must be an integer or floating-point type.

For T32 state, the compiler can use R0-R7.

For A32 state, the compiler can use R0-R12, or R14.

h

Operand must be an integer or floating-point type.

For T32 state, the compiler can use R8-R12, or R14.

Not valid for A32 state.

w

Operand must be a floating-point or vector type, or a 64-bit integer.

The compiler can use S0-S31, D0-D31, or Q0-Q15, depending on the size of the operand type.

t

Operand must be an integer or floating-point type.

The compiler can use S0-S31, D0-D15, or Q0-Q7.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Te

Operand must be an integer or 32-bit floating-point type.

The compiler can use an even numbered general purpose register in the range R0-R14.

To

Operand must be an integer or 32-bit floating-point type.

The compiler can use an odd numbered general purpose register in the range R1-R11.

The compiler never selects a register that is not available for register allocation. Similarly, R9 is reserved when compiling with `-frwpi`, and is not selected. The compiler might also reserve one or two registers to use as a frame pointer and a base pointer. The number of registers available for inline assembly operands therefore might be less than the number implied by the ranges above. This number might also vary with the optimization level.

If you use a 64-bit value as an operand to an inline assembly statement in A32 or 32-bit T32 instructions, and you use the `r` constraint code, then an even/odd pair of general purpose registers is allocated to hold it. This register allocation is not guaranteed for the `l` or `h` constraints.

Using the `r` constraint code enables the use of instructions like `LDREXD` and `STREXD`, which require an even/odd register pair. You can reference the registers holding the most and least significant halves of the value with the `Q` and `R` template modifiers. For an example of using template modifiers, see Template modifiers for AArch32 state.

Constants

The constant constraints accept different ranges depending on the selected instruction set. These ranges correspond to the ranges of immediate operands that are available for the different instruction sets. You can use them with a register constraint to write inline assembly that emits optimal code for multiple architectures without having to change the assembly code. The emitted code uses immediate operands when possible. See Using multiple alternative operand constraints for more information.

Constraint code	16-bit T32 instructions	32-bit T32 instructions	A32 instructions
`j`	Invalid immediate constraint.	An immediate integer between 0 and 65535 (valid for `MOVW`).	An immediate integer between 0 and 65535 (valid for `MOVW`).
`I`	[0, 255]	Modified immediate value for 32-bit T32 instructions.	Modified immediate value for A32 instructions.
`J`	[-255, -1]	[-4095, 4095]	[-4095, 4095]
`K`	8-bit value shifted left any amount.	Bitwise inverse of a modified immediate value for a 32-bit T32 instruction.	Bitwise inverse of a modified immediate value for an A32 instruction.
`L`	[-7, 7]	Arithmetic negation of a modified immediate value for a 32-bit T32 instruction.	Arithmetic negation of a modified immediate value for an A32 instruction.
`N`	An immediate integer between 0 and 31.	Invalid immediate constraint.	Invalid immediate constraint.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Constraint code	16-bit T32 instructions	32-bit T32 instructions	A32 instructions
o	An immediate integer which is a multiple of 4 between -508 and 508.	Invalid immediate constraint.	Invalid immediate constraint.

2.8.4.5 Constraint codes for AArch64 state

The following constraint codes are specific to AArch64 state.

Registers

r

The compiler can use a 64-bit general purpose register, X0-X30.

If you want the compiler to use the 32-bit general purpose registers W0-W30 instead, use the `w` template modifier.


Note

Most A64 integer instructions can operate on either 32-bit or 64-bit registers. The register width is determined by the register identifier, where W<n> means 32-bit and X<n> means 64-bit. The names W and X, where <n> is in the range 0-30, refer to the same register. When you use the 32-bit form of an instruction, the upper 32 bits of the source registers are ignored and the upper 32 bits of the destination register are set to zero.

w

The compiler can use a SIMD or floating-point register, V0-V31.

The `b`, `h`, `s`, `d`, and `q` template modifiers can override this behavior.

x

Operand must be a 128-bit vector type.

The compiler can use a low SIMD register, V0-V15 only.

Constants

z

A constant with value zero, printed as the zero register XZR or WZR. Useful when combined with `r` (see Using multiple alternative operand constraints) to represent an operand that can be either a general-purpose register or the zero register.

I

[0, 4095], with an optional left shift by 12. The range that the `ADD` and `SUB` instructions accept.

J

[-4095, 0], with an optional left shift by 12.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

K

An immediate that is valid for 32-bit logical instructions. For example, `AND`, `ORR`, and `EOR`.

L

An immediate that is valid for 64-bit logical instructions. For example, `AND`, `ORR`, and `EOR`.

M

An immediate that is valid for a `MOV` instruction with a destination of a 32-bit register. Valid values are all values that the `K` constraint accepts, plus the values that the `MOVZ`, `MOVN`, and `MOVK` instructions accept.

N

An immediate that is valid for a `MOV` instruction with a destination of a 64-bit register. Valid values are all values that the `L` constraint accepts, plus the values that the `MOVZ`, `MOVN`, and `MOVK` instructions accept.

Related information

Inline assembly template modifiers on page 418

2.8.4.6 Using multiple alternative operand constraints

There are many instructions that can take either an immediate value with limited range or a register as one of their operands.

To generate optimal code for an instruction, use the immediate version of the instruction where possible. Using an immediate value avoids needing a register to hold the operand, and any extra instructions to load the operand into that register. However, you can only use an immediate value if the operand is a compile-time constant, and is in the appropriate range.

To generate the best possible code, you can provide multiple constraint codes for an operand. The compiler selects the most restrictive one that it can use.

Example: Use of the "Ir" constraint string in inline assembly for an add() function

```
int add(int a, int b) {
  int r;
  // Here, the "Ir" constraint string tells the compiler that operand b can be
  // an immediate, but if it is not a constant, or not in the appropriate
  // range for an arithmetic instruction, it can be placed in a register.
  __asm("add %[r], %[a], %[b]"
      : [r] "=r" (r)
      : [a] "r" (a),
        [b] "Ir" (b));
  return r;
}

// At -O2 or higher, the call to add is inlined and optimised, so that the
// immediate form of the add instruction can be used.
int add_42(int a) {
  return add(a, 42);
}

// Here, the immediate cannot be used by the add instruction, so the compiler
// emits a movw instruction to load the value 12345 into a register.
int add_12345(int a) {
  return add(a, 12345);
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
}
```

2.8.5 Inline assembly template modifiers

Template modifiers are characters that you can insert into the assembly string, between the `%` character and the name or number of an operand reference.

For example, `%c1`, where `c` is the template modifier, and `1` is the number of the operand reference. Template modifiers change the way that the operand is printed in the string. This change is sometimes required so the operand is in the form that some instructions or directives expect.

2.8.5.1 Template modifiers common to AArch32 state and AArch64 state

A list of the template modifiers that are common to both AArch32 state and AArch64 state.

These modiifiers are:

c

Valid for an immediate operand. Prints it as a plain value without a preceding `#`. Use this template modifier when using the operand in `.word`, or another data-generating directive, which needs an integer without the `#`.

n

Valid for an immediate operand. Prints the arithmetic negation of the value without a preceding `#`.

Example: Use of the 'c' template modifier

```
// This uses an operand as the value in the .word directive. The .word
// directive does not accept numbers with a preceeding #, so we use the 'c'
// template modifier to print just the value.
int foo() {
  int val;
  __asm (R"(
      ldr %0, 1f
      b 2f
    1:
      .word %c1
    2:
    )"
    : "=r" (val)
    : "i" (0x12345678));
  return val;
}
```

2.8.5.2 Template modifiers for AArch32 state

A list of the template modifiers that are specific to AArch32 state.

These modiifiers are:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

a

If the operand uses a register constraint, it is printed surrounded by square brackets. If it uses a constant constraint, it is printed as a plain immediate, with no preceding `#`.

y

The operand must be a 32-bit floating-point type, using a register constraint. It is printed as the equivalent D register with an index. For example, the register S2 is printed as `d1[0]`, and the register S3 is printed as `d1[1]`.

B

The operand must use a constant constraint, and is printed as the bitwise inverse of the value, without a preceding `#`.

L

The operand must use a constant constraint, and is printed as the least-significant 16 bits of the value, without a preceding `#`.

Q

The operand must use the `r` constraint, and must be a 64-bit integer or floating-point type. The operand is printed as the register holding the least-significant half of the value.

R

The operand must use the `r` constraint, and must be a 64-bit integer or floating-point type. The operand is printed as the register holding the most-significant half of the value.

H

The operand must use the `r` constraint, and must be a 64-bit integer or floating-point type. The operand is printed as the highest-numbered register holding half of the value.

e

The operand must be a 128-bit vector type, using the `w` or `x` constraint. The operand is printed as the D register that overlaps the low half of the allocated Q register.

f

The operand must be a 128-bit vector type, using the `w` or `x` constraint. The operand is printed as the D register that overlaps the high half of the allocated Q register.

m

The operand must use the `m` constraint, and is printed as a register without the surrounding square brackets.

Example: Use of the 'Q' and 'R' template modifiers in AArch32 state

```
// In AArch32 state, the 'Q' and 'R' template modifiers can be used to print
// the registers holding the least- and most-significant half of a 64-bit
// operand.
uint64_t atomic_swap(uint64_t new_val, uint64_t *addr) {
  uint64_t old_val;
  unsigned temp;
  __asm volatile(
       "dmb ish\n"
       "1:\n"
       "ldrexd %Q[old], %R[old], %[addr]\n"
       "strexd %[temp], %Q[new], %R[new], %[addr]\n"
       "cmp %[temp], #0\n"
       "bne 1b\n"
       "dmb ish\n"
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    : [new] "+&r" (old_val),
      [temp] "=&r" (temp)
    : [old] "r"   (new_val),
      [addr] "m"  (*addr));
  return old_val;
}
```

2.8.5.3 Template modifiers for AArch64 state

A list of the template modifiers that are specific to AArch64 state.

In AArch64 state, register operands are printed as `x` registers for integer types and `v` registers for floating-point and vector types by default. You can use the template modifiers to override this behavior.

The modiifiers are:

a
: Operand constraint must be `r`. Prints the register name surrounded by square brackets. Suitable for use as a memory operand.

w
: Operand constraint must be `r`. Prints the register using its 32-bit `w` name.

x
: Operand constraint must be `r`. Prints the register using its 64-bit `x` name.

b
: Operand constraint must be `w` or `x`. Prints the register using its 8-bit `b` name.

h
: Operand constraint must be `w` or `x`. Prints the register using its 16-bit `h` name.

s
: Operand constraint must be `w` or `x`. Prints the register using its 32-bit `s` name.

d
: Operand constraint must be `w` or `x`. Prints the register using its 64-bit `d` name.

q
: Operand constraint must be `w` or `x`. Prints the register using its 128-bit `q` name.

Example: Use of the 's' template modifier in AArch64 state

```
// In AArch64 state, the 's' template modifier causes these operands to be
// printed as S registers, instead of the default of V registers.
float add(float a, float b) {
  float result;
  __asm("fadd %s0, %s1, %s2"
    : "=w" (result)
    : "w"  (a), "w"  (b));
  return result;
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.8.6 Forcing inline assembly operands into specific registers

Sometimes specifying the exact register that is used for an operand is preferable to letting the compiler allocate a register automatically.

For example, the inline assembly block might contain a call to a function or system call that expects an argument or return value in a particular register.

To specify the register to use, the operand of the inline assembly statement must be a local register variable, which you declare as follows:

```
register int foo __asm("r2");
register float bar __asm("s4") = 3.141;
```

A local register variable is guaranteed to be held in the specified register in an inline assembly statement where it is used as an operand. Elsewhere it is treated as a normal variable, and can be stored in any register or in memory. Therefore, a function can contain multiple local register variables that use the same register if only one local register variable is in any single inline assembly statement.

Example: A function that uses named register variables to make a Linux read system call

```
// This function uses named register variables to make a Linux 'read' system call.
// The three arguments to the system call are held in r0-r2, and the system
// call number is placed in r7.
int syscall_read(register int fd, void *buf, unsigned count) {
  register unsigned r0 __asm("r0") = fd;
  register unsigned r1 __asm("r1") = buf;
  register unsigned r2 __asm("r2") = count;
  register unsigned r7 __asm("r7") = 0x900003;
  __asm("svc #0"
      : "+r" (r0)
      : "r"  (r1), "r" (r2), "r" (r7));
  return r0;
}
```

2.8.7 Symbol references and branches into and out of inline assembly

Symbols that are defined in an inline assembly statement can only be referred to from the same inline assembly statement.

The compiler can optimize functions containing inline assembly, which can result in the removal or duplication of the inline assembly statements. To define symbols in assembly and use them elsewhere, use file-scope inline assembly, or a separate assembly language source file.

With the exception of function calls, it is not permitted to branch out of an inline assembly block, including branching to other inline assembly blocks. The optimization passes of the compiler assume that inline assembly statements only exit by reaching the end of the assembly block, and optimize the surrounding function accordingly.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

It is valid to call a function from inside inline assembly, as that function returns control-flow back to the inline assembly code.

We do not recommend directly referencing global data or functions from inside an assembly block by using their names in the assembly string. Often such references appear to work, but the compiler does not know about the reference.

If the global data or functions are only referenced inside inline assembly statements, then the compiler might remove these global data or functions.

To prevent the compiler from removing global data or functions which are referenced from inline assembly statements, you can:

- Use `__attribute__((used))` with the global data or functions.
- Pass the reference to global data or functions as operands to inline assembly statements.

We recommend passing the reference to global data or functions as operands to inline assembly statements so that if the final image does not require the inline assembly statements and the referenced global data or function, then they can be removed.

Example: Symbol references and branches into and out of inline assembly

```
static void test1(void) { /* ... */ }

// This function attempts to call the function foo from inside inline assembly.
// In some situations this might appear to work, but if foo is not referenced
// anywhere else (including if all calls to it from C got inlined), the
// compiler could remove the definition of foo, so this would fail to link.
void test2() {
  __asm volatile(
      "bl test1"
    : /* no outputs */
    : /* no inputs */
    : "r0", "r1", "r2", "r3", "r12", "lr");
}

// This function is the same as test2, except that it passes a reference to test1
// into the inline assembly as an operand. This lets the compiler know about the
// reference, so the definition of test1 is not removed (unless, the
// definition of test2_fixed can also be removed). In C++, this has the
// additional advantage that the operand uses the source name of the function,
// not the mangled name (_ZL3test1v) which would have to be used if writing the
// symbol name directly in the assembly string.
void test2_fixed() {
  __asm volatile(
      "bl %[test1]"
    : /* no outputs */
    : [test1] "i" (test1)
    : "r0", "r1", "r2", "r3", "r12", "lr");
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2.8.8 Duplication of labels in inline assembly statements

You can use labels inside inline assembly, for example as the targets of branches or PC-relative load instructions. However, you must ensure that the labels that you create are valid if the compiler removes or duplicates an inline assembly statement.

Duplication can happen when a function containing an inline assembly statement is inlined in multiple locations. Removal can happen if an inline assembly statement is not reachable, or its result is unused and it has no side-effects.

If regular labels are used inside inline assembly, then duplication of the assembly could lead to multiple definitions of the same symbol, which is invalid. Multiple definitions can be avoided either by using Numeric local labels, or using the `%=` template string. The `%=` template string is expanded to a number that is unique to each instance of an inline assembly statement. Duplicated statements have different numbers. All uses of `%=` in an instance of the inline assembly statement have the same value. This allows you to create label names that can be referenced in the same inline assembly statement, but which do not conflict with other copies of the same statement.


Note

Unique numbers from the `%=` template string might still result in the creation of duplicate labels. For example, label names `loop%=` and `loop1%=` can result in duplicate labels. The label for instance number 0 of `loop1%=` evaluates to `loop10`. The label for instance number 10 of `loop%=` also evaluates to `loop10`.

To avoid such duplicate labels, choose the label names carefully.

Example: How to use the %= template string

Create the file `memcpy.c` containing:

```
#include <assert.h>

__attribute__((always_inline)) void memcpy_words(int *src, int *dst, int len)
{
  assert((len % 4) == 0);
  int tmp;

  // This uses the "%=" template string to create a label which can be used
  // elsewhere inside the assembly block, but which will not conflict with
  // inlined copies of it.
  __asm
  (
    ".Lloop%=:\n\t"
      "ldr %[tmp], %[src], #4\n\t"
      "str %[tmp], %[dst], #4\n\t"
      "subs %[len], #4\n\t"
      "bne .Lloop%="

    : [dst] "=&m" (*dst),
      [tmp] "=&r" (tmp),
      [len] "+r" (len)
    : [src] "m" (*src)
  );
}

void do_memcpy_words(int *src, int *dst, int len)
{
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
  memcpy_words(src, dst, len);
}
```

Build the example using the following command:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -O1 -c -S memcpy.c -o memcpy.s
```

The compiler uses `.Lloop0` for instances of `.Lloop%=` within `memcpy_words()` itself, and uses `.Lloop1` for instances within the inlined copy inside `do_memcpy_words()`. If you build the example, you can see the result in the output assembly file `memcpy.s`:

```
...
memcpy_words:
...
    @APP
.Lloop0:
    ldr    r3, [r0], #4
    str    r3, [r1], #4
    subs   r2, #4
    bne    .Lloop0
    @NO_APP
    bx     lr
...
do_memcpy_words:
...
    @APP
.Lloop1:
    ldr    r3, [r0], #4
    str    r3, [r1], #4
    subs   r2, #4
    bne    .Lloop1
    @NO_APP
    bx     lr
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3. armlink Reference

A list of the command-line options for the `armlink` command, and other reference information that is relevant to `armlink`.

3.1 armlink Command-line Options

Describes the command-line options supported by the Arm linker, `armlink`.

3.1.1 --any_contingency

Permits extra space in any execution regions containing `.ANY` sections for linker-generated content such as veneers and alignment padding.

Syntax

```
--any_contingency
```

Parameters

None.

Operation

Two percent of the extra space in such execution regions is reserved for veneers.

When a region is about to overflow because of potential padding, `armlink` lowers the priority of the `.ANY` selector.

This option is off by default. That is, `armlink` does not attempt to calculate padding and strictly follows the `.ANY` priorities.

Use this option with the `--scatter` option.

Related information

--info=topic[,topic,...] (armlink) on page 474
--any_sort_order=order on page 427
--scatter=filename on page 529
Behavior when .ANY sections overflow because of linker-generated content on page 681

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.2 --any_placement=algorithm

Controls the placement of sections that are placed using the `.ANY` module selector.

Default

The default option is `--any_placement=worst_fit`.

Syntax

```
--any_placement=<algorithm>
```

Parameters

`<algorithm>`

One of the following:

`best_fit`

Place the section in the execution region that currently has the least free space but is also sufficient to contain the section.

`first_fit`

Place the section in the first execution region that has sufficient space. The execution regions are examined in the order they are defined in the scatter file.

`next_fit`

Place the section using the following rules:

- Place in the current execution region if there is sufficient free space.
- Place in the next execution region only if there is insufficient space in the current region.
- Never place a section in a previous execution region.

`worst_fit`

Place the section in the execution region that currently has the most free space.

Use this option with the `--scatter` option.

Operation

The placement algorithms interact with scatter files and `--any_contingency` as follows:

Interaction with normal scatter-loading rules

Scatter-loading with or without `.ANY` assigns a section to the most specific selector. All algorithms continue to assign to the most specific selector in preference to `.ANY` priority or size considerations.

Interaction with .ANY priority

Priority is considered after assignment to the most specific selector in all algorithms.

`worst_fit` and `best_fit` consider priority before their individual placement criteria. For example, you might have `.ANY1` and `.ANY2` selectors, with the `.ANY1` region having the most

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

free space. When using `worst_fit` the section is assigned to `.ANY2` because it has higher priority. Only if the priorities are equal does the algorithm come into play.

`first_fit` considers the most specific selector first, then priority. It does not introduce any more placement rules.

`next_fit` also does not introduce any more placement rules. If a region is marked full during `next_fit`, that region cannot be considered again regardless of priority.

Interaction with --any_contingency

The priority of a `.ANY` selector is reduced to `0` if the region might overflow because of linker-generated content. This is enabled and disabled independently of the sorting and placement algorithms.

`armlink` calculates a worst-case contingency for each section.

For `worst_fit`, `best_fit`, and `first_fit`, when a region is about to overflow because of the contingency, `armlink` lowers the priority of the related `.ANY` selector.

For `next_fit`, when a possible overflow is detected, `armlink` marks that section as `FULL` and does not consider it again. This stays consistent with the rule that when a section is full it can never be revisited.

Related information

--any_sort_order=order on page 427
--info=topic[,topic,...] (armlink) on page 474
--scatter=filename on page 529
Examples of using placement algorithms for .ANY sections on page 675
Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 677
--any_contingency on page 425
Manual placement of unassigned sections on page 670
Syntax of an input section description on page 730
Behavior when .ANY sections overflow because of linker-generated content on page 681

3.1.3 --any_sort_order=order

Controls the sort order of input sections that are placed using the `.ANY` module selector.

Default

The default option is `--any_sort_order=descending_size`.

Syntax

```
--any_sort_order=<order>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<order>

One of the following:

descending_size

Sort input sections in descending size order.

cmdline

The order that the section appears on the linker command-line. The command-line order is defined as `File.Object.Section` where:

- `Section` is the section index, `sh_idx`, of the `Section` in the `Object`.
- `Object` is the order that `Object` appears in the `File`.
- `File` is the order the `File` appears on the command line.

The order the `Object` appears in the `File` is only significant if the file is an `ar` archive.

By default, sections that have the same properties are resolved using the creation index. The `--tiebreaker` command-line option does not have any effect in the context of `--any_sort_order`.

Use this option with the `--scatter` option.

Operation

The sorting governs the order that sections are processed during `.ANY` assignment. Normal scatter-loading rules, for example `RO` before `RW`, are obeyed after the sections are assigned to regions.

Related information

--info=topic[,topic,...] (armlink) on page 474
--scatter=filename on page 529
--any_contingency on page 425
Manual placement of unassigned sections on page 670
Examples of using sorting algorithms for .ANY sections on page 679

3.1.4 --api, --no_api

Enables and disables API section sorting. API sections are the sections that are called the most within a region.

Default

The default is `--no_api`. The linker automatically switches to `--api` if at least one execution region contains more code than the smallest inter-section branch. The smallest inter-section branch depends on the code in the region and the target processor:

128MB

Execution region contains only A64 instructions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

32MB

Execution region contains only A32 instructions.

16MB

Execution region contains 32-bit T32 instructions.

4MB

Execution region contains only 16-bit T32 instructions.

Syntax

```
--api
--no_api
```

Parameters

None.

Operation

In large region mode the API sections are extracted from the region and then inserted closest to the hotspots of the calling sections. This minimizes the number of veneers generated.

Related information

--largeregions, --no_largeregions on page 487
Linker-generated veneers on page 597

3.1.5 --autoat, --no_autoat

Controls the automatic assignment of `__at` sections to execution regions.

Default

The default is `--autoat`.

Syntax

```
--autoat
--no_autoat
```

Parameters

None.

Restrictions

You cannot use `__at` section placement with position independent execution regions.

If you use `__at` sections with overlays, you cannot use `--autoat` to place those sections. You must specify the names of `__at` sections in a scatter file manually, and specify the `--no_autoat` option.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

`__at` sections are sections that must be placed at a specific address.

If enabled, the linker automatically selects an execution region for each `__at` section. If a suitable execution region does not exist, the linker creates a load region and an execution region to contain the `__at` section.

If disabled, the standard scatter-loading section selection rules apply.

Related information

Syntax of a scatter file on page 711
Placement of __at sections at a specific address on page 663
Automatic placement of __at sections on page 664
Manual placement of __at sections on page 666

3.1.6 --bare_metal_pie

Specifies the bare-metal *Position Independent Executable* (PIE) linking model.

Default

The following default settings are automatically specified:

- `--fpic`.
- `--pie`.
- `--ref_pre_init`.

Syntax

```
--bare_metal_pie
```

Parameters

None.

Related information

--fpic on page 469
--pie on page 515
--ref_pre_init, --no_ref_pre_init on page 522
Bare-metal Position Independent Executables

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.7 --bare_metal_sysv

Enables the use of scatter files to place the stack and heap in a SysV bare-metal context.

Syntax

```
--bare_metal_sysv
```

Parameters

None.

Operation

Use with `--sysv` to allow the the use of the region names `ARM_LIB_STACK`, `ARM_LIB_HEAP`, and `ARM_LIB_STACKHEAP` in a scatter file.

Related information

--shared on page 532
--sysv on page 548

3.1.8 --be8

Specifies byte-invariant addressing big-endian (BE-8) mode.

Syntax

```
--be8
```

Parameters

None.

Operation

BE-8 is the default byte addressing mode for Arm®v6 and later big-endian images. As a result, the linker reverses the endianness of the instructions to give little-endian code and big-endian data for input objects that are compiled or assembled as big-endian.

Byte-invariant addressing mode is only available on Arm processors that support architecture Armv6 and later.

3.1.9 --be32

Specifies word-invariant addressing big-endian (BE-32) mode.

Syntax

```
--be32
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

This option produces big-endian code and data.

By default, big-endian mode uses byte-invariant addressing, `--be8`.

Example

```
armlink --cpu=7-R --be32 -o test.axf test.o
```

Related information

--be8 on page 431

3.1.10 --bestdebug, --no_bestdebug

Selects between linking for smallest code and data size or for best debug illusion.

Default

The default is `--no_bestdebug`. The smallest COMDAT groups are selected when linking, at the expense of a possibly slightly poorer debug illusion.

Syntax

```
--bestdebug
--no_bestdebug
```

Parameters

None.

Operation

Input objects might contain common data (COMDAT) groups, but these might not be identical across all input objects because of differences such as objects compiled with different optimization levels.

Use `--bestdebug` to select COMDAT groups with the best debug view. Be aware that the code and data of the final image might not be the same when building with or without debug.

Example

For two objects compiled with different optimization levels:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c -O2 file1.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c -O0 file2.c
armlink --bestdebug file1.o file2.o -o image.axf
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

-o filename, --output=filename (armlink) on page 509
Elimination of common section groups on page 610
Elimination of unused sections on page 611

3.1.11 --blx_arm_thumb, --no_blx_arm_thumb

Enables the linker to use the BLX instruction for A32 to T32 state changes.

Syntax

```
--blx_arm_thumb
--no_blx_arm_thumb
```

Parameters

None.

Operation

If the linker cannot use BLX it must use an A32 to T32 interworking veneer to perform the state change.

This option is on by default. It has no effect if the target architecture does not support BLX or when linking for AArch64 state.

Related information

--blx_thumb_arm, --no_blx_thumb_arm on page 433

3.1.12 --blx_thumb_arm, --no_blx_thumb_arm

Enables the linker to use the BLX instruction for T32 to A32 state changes.

Syntax

```
--blx_thumb_arm
--no_blx_thumb_arm
```

Parameters

None.

Operation

If the linker cannot use BLX it must use a T32 to A32 interworking veneer to perform the state change.

This option is on by default. It has no effect if the target architecture does not support BLX or when linking for AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--blx_arm_thumb, --no_blx_arm_thumb on page 433

3.1.13 --branchnop, --no_branchnop

Enables or disables the replacement of any branch with a relocation that resolves to the next instruction with a NOP.

Default

The default is `--branchnop`.

Syntax

```
--branchnop
--no_branchnop
```

Parameters

None.

Operation

The default behavior is to replace any branch with a relocation that resolves to the next instruction with a NOP. However, there are cases where you might want to use `--no_branchnop` to disable this behavior. For example, when performing verification or pipeline flushes.

Related information

--inline, --no_inline on page 479
--tailreorder, --no_tailreorder on page 549
About branches that optimize to a NOP on page 618

3.1.14 --callgraph, --no_callgraph

Creates a file containing a static callgraph of functions.

Default

The default is `--no_callgraph`.

Syntax

```
--callgraph
--no_callgraph
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

The callgraph gives definition and reference information for all functions in the image.


Note

If you use the `--partial` option to create a partially linked object, then no callgraph file is created.


Note

We do not recommend using the `--callgraph` option with the `armclang` option `-fno-omit-frame-pointer` to measure the stack usage of the function. Local variables on the stack that are accessed through the frame pointer are not counted in the size of the stack frame. Therefore, `armlink` underestimates the amount of stack used.

The callgraph file:

- Is saved in the same directory as the generated image.
- Has the name of the linked image with the extension, if any, replaced by the callgraph output extension, either `.htm` or `.txt`. Use the `--callgraph_file=<filename>` option to specify a different callgraph filename.
- Has a default output format of HTML. Use the `--callgraph_output=<fmt>` option to control the output format.


Note

If the linker is to calculate the function stack usage, any functions defined in the assembler files must have the appropriate:

- `.cfi_startproc` and `.cfi_endproc` directives.
- `.cfi_sections .debug_frame` directive.

The linker lists the following for each function `func`:

- Instruction set state for which the function is compiled (A32, T32, or A64).
- Set of functions that call `func`.
- Set of functions that are called by `func`.
- Number of times the address of `func` is used in the image.

In addition, the callgraph identifies functions that are:

- Called through interworking veneers.
- Defined outside the image.
- Permitted to remain undefined (weak references).
- Called through a *Procedure Linkage Table* (PLT).
- Not called but still exist in the image.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The static callgraph also gives information about stack usage. It lists the:

- Size of the stack frame used by each function.
- Maximum size of the stack used by the function over any call sequence, that is, over any acyclic chain of function calls.

If there is a cycle, or if the linker detects a function with no stack size information in the call chain, `+ Unknown` is added to the stack usage. A reason is added to indicate why stack usage is unknown.

The linker reports missing stack frame information if there is no debug frame information for the function.

For indirect functions, the linker cannot reliably determine which function made the indirect call. This might affect how the maximum stack usage is calculated for a call chain. The linker lists all function pointers used in the image.

Use frame directives in assembly language code to describe how your code uses the stack. These directives ensure that debug frame information is present for debuggers to perform stack unwinding or profiling.

Related information

--callgraph_file=filename on page 436
--callgraph_output=fmt on page 437
--callgraph_subset=symbol[,symbol,...] on page 437
--cgfile=type on page 438
--cgsymbol=type on page 439
--cgundefined=type on page 439
-fomit-frame-pointer, -fno-omit-frame-pointer on page 77

3.1.15 --callgraph_file=filename

Controls the output filename of the callgraph.

Syntax

```
--callgraph_file=<filename>
```

Parameters

`<filename>`

The callgraph filename.

Operation

The default filename is the name of the linked image with the extension, if any, replaced by the callgraph output extension, either `.htm` or `.txt`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--callgraph, --no_callgraph on page 434
--callgraph_output=fmt on page 437
--callgraph_subset=symbol[,symbol,...] on page 437
--cgfile=type on page 438
--cgsymbol=type on page 439
--cgundefined=type on page 439

3.1.16 --callgraph_output=fmt

Controls the output format of the callgraph.

Default

The default is `--callgraph_output=html`.

Syntax

```
--callgraph_output=<fmt>
```

Parameters

<fmt>

One of the following:

html

Outputs the callgraph in HTML format.

text

Outputs the callgraph in plain text format.

Related information

--callgraph, --no_callgraph on page 434
--callgraph_file=filename on page 436
--callgraph_subset=symbol[,symbol,...] on page 437
--cgfile=type on page 438
--cgsymbol=type on page 439
--cgundefined=type on page 439

3.1.17 --callgraph_subset=symbol[,symbol,...]

Creates a file containing a static callgraph for one or more specified symbols.

Syntax

```
--callgraph_subset=<symbol>[,<symbol>,...]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<symbol>

A comma-separated list of symbols.

Operation

The callgraph file:

- Is saved in the same directory as the generated image.
- Has the name of the linked image with the extension, if any, replaced by the callgraph output extension, either `.htm` or `.txt`. Use the `--callgraph_file=<filename>` option to specify a different callgraph filename.
- Has a default output format of HTML. Use the `--callgraph_output=<fmt>` option to control the output format.

Related information

--callgraph, --no_callgraph on page 434
--callgraph_file=filename on page 436
--callgraph_output=fmt on page 437
--cgfile=type on page 438
--cgsymbol=type on page 439
--cgundefined=type on page 439

3.1.18 --cgfile=type

Controls the type of files to use for obtaining the symbols to be included in the callgraph.

Default

The default is `--cgfile=all`.

Syntax

```
--cgfile=<type>
```

Parameters

<type>

One of the following:

all

Includes symbols from all files.

user

Includes only symbols from user defined objects and libraries.

system

Includes only symbols from system libraries.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--callgraph, --no_callgraph on page 434
--callgraph_file=filename on page 436
--callgraph_output=fmt on page 437
--callgraph_subset=symbol[,symbol,...] on page 437
--cgsymbol=type on page 439
--cgundefined=type on page 439

3.1.19 --cgsymbol=type

Controls what symbols are included in the callgraph.

Default

The default is `--cgsymbol=all`.

Syntax

```
--cgsymbol=<type>
```

Parameters

<type>

One of the following:

all

Includes both local and global symbols.

locals

Includes only local symbols.

globals

Includes only global symbols.

Related information

--callgraph, --no_callgraph on page 434
--callgraph_file=filename on page 436
--callgraph_output=fmt on page 437
--callgraph_subset=symbol[,symbol,...] on page 437
--cgfile=type on page 438
--cgundefined=type on page 439

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.20 --cgundefined=type

Controls what undefined references are included in the callgraph.

Default

The default is `--cgundefined=all`.

Syntax

```
--cgundefined=<type>
```

Parameters

<type>

One of the following:

all

Includes both function entries and calls to undefined weak references.

entries

Includes function entries for undefined weak references.

calls

Includes calls to undefined weak references.

none

Omits all undefined weak references from the output.

Related information

--callgraph, --no_callgraph on page 434
--callgraph_file=filename on page 436
--callgraph_output=fmt on page 437
--callgraph_subset=symbol[,symbol,...] on page 437
--cgfile=type on page 438
--cgsymbol=type on page 439

3.1.21 --check_pac_mismatch

Causes `armlink` to report a warning when linking PAC with non-PAC objects.

Syntax

```
--check_pac_mismatch
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

When linking PAC with non-PAC objects, `armlink` reports no diagnostic message by default.

With the `--check_pac_mismatch` option, `armlink` outputs the warning:

```
Warning: L6142W: Composition of PAC and non-PAC objects detected. Use --info=pac to
print out the list of objects with their corresponding PAC mark.
```

Use the `--info=pac` option to report a list of the PAC and non-PAC objects that are required to link the image.

Related information

--info=topic[,topic,...] (armlink) on page 474
--require-bti on page 524
--library_security=protection on page 492

3.1.22 --comment_section, --no_comment_section

Controls the inclusion of a comment section `.comment` in the final image.

Default

The default is `--comment_section`.

Syntax

```
--comment_section
--no_comment_section
```

Parameters

None.

Operation

Use `--no_comment_section` to remove the `.comment` section, to help reduce the image size.



Note

You can also use the `--filtercomment` option to merge comments.

Related information

Linker merging of comment sections on page 619
--filtercomment, --no_filtercomment on page 465

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.23 --cppinit, --no_cppinit

Enables the linker to use alternative C++ libraries with a different initialization symbol if required.

Default

The default for `<symbol>` is `__cpp_initialize__aeabi_`.

Syntax

```
--cppinit=<symbol>
--no-cppinit
```

Parameters

`<symbol>`

The initialization symbol to use.

Operation

The linker adds a non-weak reference to `<symbol>` if any static constructor or destructor sections are detected.

For `--cppinit=__cpp_initialize__aeabi_` in AArch32 state, the linker processes R_ARM_TARGET1 relocations as R_ARM_REL32, because this is required by the `__cpp_initialize__aeabi_` function. In all other cases R_ARM_TARGET1 relocations are processed as R_ARM_ABS32.



Note

There is no equivalent of R_ARM_TARGET1 in AARCH64 state.

`--no_cppinit` does not take a `<symbol>` argument and means do not add a reference.

Related information

--preinit, --no_preinit on page 519
--ref_cpp_init, --no_ref_cpp_init on page 521

3.1.24 --cpu=list (armlink)

Lists the architecture and processor names that are supported by the `--cpu=name` option.

Syntax

```
--cpu=list
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

`list`

List the supported architecture and processor names.

Restrictions

Architectures and processors for Arm®v8.4-A or later are not listed because you cannot use `--cpu=<name>` for such targets.

Related information

--cpu=name (armlink) on page 443
--fpu=list (armlink) on page 469
--fpu=name (armlink) on page 470

3.1.25 --cpu=name (armlink)

Enables code generation for the selected Arm® processor or architecture.

Syntax

To specify the processor or architecture:

```
--cpu=<name>
```

To list all supported architecture and processor names:

```
--cpu=list
```

Parameters

`<name>`

The name of a processor or architecture:

- Processor and architecture names are not case-sensitive.
- Wildcard characters are not accepted.

The following table shows the supported architectures.

Table 3-1: Supported Arm architectures

Architecture name	Description
`6-M`	Armv6 architecture microcontroller profile.
`6S-M`	Armv6 architecture microcontroller profile with OS extensions.
`7-A`	Armv7 architecture application profile.
`7-A.security`	Armv7-A architecture profile with the Security Extension and includes the `SMC` instruction (formerly `SMI`).
`7-R`	Armv7 architecture real-time profile.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Architecture name	Description
`7-M`	Armv7 architecture microcontroller profile.
`7E-M`	Armv7-M architecture profile with DSP extension.
`8-A.32`	Armv8-A architecture profile, AArch32 state.
`8-A.32.crypto`	Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8-A.32.no_neon`	Armv8-A architecture profile, AArch32 state without Advanced SIMD instructions.
`8-A.64`	Armv8-A architecture profile, AArch64 state.
`8-A.64.crypto`	Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.1-A.32`	Armv8.1, for Armv8-A architecture profile, AArch32 state.
`8.1-A.32.crypto`	Armv8.1, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.1-A.64`	Armv8.1, for Armv8-A architecture profile, AArch64 state.
`8.1-A.64.crypto`	Armv8.1, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.2-A.32`	Armv8.2, for Armv8-A architecture profile, AArch32 state.
`8.2-A.32.crypto`	Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.2-A.32.crypto.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions and the `VSDOT` and `VUDOT` instructions.
`8.2-A.32.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch32 state with the `VSDOT` and `VUDOT` instructions.
`8.2-A.64`	Armv8.2, for Armv8-A architecture profile, AArch64 state.
`8.2-A.64.crypto`	Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.2-A.64.crypto.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions and the `SDOT` and `UDOT` instructions.
`8.2-A.64.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch64 state with the `SDOT` and `UDOT` instructions.
`8.3-A.32`	Armv8.3, for Armv8-A architecture profile, AArch32 state.
`8.3-A.32.crypto`	Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.3-A.32.crypto.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions and the `VSDOT` and `VUDOT` instructions.
`8.3-A.32.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch32 state with the `VSDOT` and `VUDOT` instructions.
`8.3-A.64`	Armv8.3, for Armv8-A architecture profile, AArch64 state.
`8.3-A.64.crypto`	Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.3-A.64.crypto.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions and the `SDOT` and `UDOT` instructions.
`8.3-A.64.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch64 state with the `SDOT` and `UDOT` instructions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Architecture name	Description
`8-R`	Armv8-R architecture profile, AArch32 state.
`8-R.crypto`	Armv8-R architecture profile, AArch32 state with cryptographic instructions.
`8-R.no_neon`	Armv8-R architecture profile, AArch32 state without Advanced SIMD instructions.
`8-R.64`	Armv8-R architecture profile, AArch64 state.
`8-R.64.crypto`	Armv8-R architecture profile, AArch64 state with cryptographic instructions.
`8-M.Base`	Armv8-M baseline architecture profile. Derived from the Armv6-M architecture.
`8-M.Main`	Armv8-M mainline architecture profile. Derived from the Armv7-M architecture.
`8-M.Main.dsp`	Armv8-M mainline architecture profile with DSP extension.
`8.1-M.Main`	Armv8.1-M mainline architecture profile extension.
`8.1-M.Main.dsp`	Armv8.1-M mainline architecture profile with DSP extension.
`8.1-M.Main.mve`	Armv8.1-M mainline architecture profile with *M-profile Vector Extension* (MVE) for integer operations.
`8.1-M.Main.mve.fp`	Armv8.1-M mainline architecture profile with MVE for integer and floating-point operations.


Note

The full list of supported architectures and processors depends on your license.

Restrictions

`--cpu` has the following restrictions:

- You cannot specify both a processor and an architecture on the same command line.
- Build attribute compatibility checking is supported only for AArch32 state.
- You cannot specify targets with Armv8.4-A or later architectures on the `armlink` command-line. To link for such targets, you must not specify the `--cpu` option when invoking `armlink` directly.

Operation

If you omit `--cpu`, the linker auto-detects the processor or architecture from the combination of the input object files.

Specify `--cpu=list` to list the supported processor and architecture names that you can use with `--cpu=name`.

When you include `--cpu=<name>`, `armlink`:

- Faults any input object files that rely on features that are incompatible with the specified processor or architecture.
- For library selection, acts as if at least one input object is compiled with `--cpu=<name>`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The linker also uses this option to optimize the choice of system libraries and any veneers that have to be generated when building the final image.

Related information

--cpu=list (armlink) on page 442
--fpu=list (armlink) on page 469
--fpu=name (armlink) on page 470

3.1.26 --crosser_veneershare, --no_crosser_veneershare

Enables or disables veneer sharing across execution regions.

Default

The default is `--crosser_veneershare`.

Syntax

```
--crosser_veneershare
--no_crosser_veneershare
```

Parameters

None.

Operation

`--crosser_veneershare` enables veneer sharing across execution regions.

`--no_crosser_veneershare` prohibits veneer sharing across execution regions.

Related information

--veneershare, --no_veneershare on page 557

3.1.27 --dangling-debug-address=address

Ensures that code removed by the `armlink` option `--remove` does not contain leftover debug information pointing to addresses of that code.

Syntax

```
--dangling-debug-address=<address>
```

Parameters

`<address>`

An out-of-scope address that `armlink` can assign to debug information related to removed code.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

You must have enough virtual address space after the address specified with `--dangling-debug-address` so that all the debug data relocated to that region safely points to nothing.

Operation

When `armlink` removes code, it resolves references to addresses in the removed range to `0x0` by default. Therefore, any debug information for that code now points to address `0x00000000`. Resolving to `0x00000000` is a problem when the target processor has a vector table at that address and you want to set a breakpoint at that address. Therefore, use `--dangling-debug-address` to specify an unused address to use to resolve references to the removed code.



Note

You could temporarily turn off the automatic removal of unused code with `--no-remove`. However, this option increases the overall code size.

Example: Effect of using --dangling-debug-address

For an example, see Example showing the effect of using --dangling-debug-address.

Related information

-ffunction-sections, -fno-function-sections on page 67
--remove, --no_remove on page 523
Dealing with leftover debug data for code and data removed by armlink

3.1.28 Example showing the effect of using --dangling-debug-address

This example shows the effect of using `--dangling-debug-address`.

For more information, see the --dangling-debug-address=address command-line option.

1. Create `f1.c`, `f2.c`, and `main.c` containing the following code:

```
// f1.c int increment(int a)
{
    return a + 1;
}

int decrement(int a)
{
    return a - 1;
}

// f2.c int multiply(int a, int b)
{
    return a * b;
}

int power(int a, int b)
{
    return a^b;
}

// main.c int increment(int);
int decrement(int);
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
int multiply(int, int);
int power(int, int);
int main()
{
    int value = 10;
    increment(value);
    power(value, 3);
}
```

2. Compile and link with the following commands:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a8 -g -c f1.c f2.c main.c
armlink f1.o f2.o main.o --dangling-debug-address 0xff --cpu cortex-a8 -o
 image1.axf
armlink f1.o f2.o main.o --cpu cortex-a8 -o image2.axf
```

3. Display the debug sections for each image with the following `fromelf` commands:

```
fromelf -g image1.axf

...
  FDE 0001d0: CIE 000088, init loc 008334, range 000018
    DW_CFA_advance_loc +0x4 = 0x008338
...
  FDE 0001e4: CIE 000088, init loc 0000ff, range 000018
    DW_CFA_advance_loc +0x4 = 0x000103
...
  FDE 0001f8: CIE 000088, init loc 0000ff, range 000020
    DW_CFA_advance_loc +0x4 = 0x000103
...
  FDE 00020c: CIE 000088, init loc 008390, range 000020
    DW_CFA_advance_loc +0x4 = 0x008394
...

fromelf -g image2.axf

...
 FDE 0001d0: CIE 000088, init loc 008334, range 000018
 DW_CFA_advance_loc +0x4 = 0x008338
...
 FDE 0001e4: CIE 000088, init loc 000000, range 000018
 DW_CFA_advance_loc +0x4 = 0x000004
...
 FDE 0001f8: CIE 000088, init loc 000000, range 000020
 DW_CFA_advance_loc +0x4 = 0x000004
...
 FDE 00020c: CIE 000088, init loc 008390, range 000020
 DW_CFA_advance_loc +0x4 = 0x008394
...
```

For the unused functions, `decrement()` and `multiply()`, `armlink` sets the location to `0x000000` as shown for `image2.axf`. If there is a vector table at address `0x00`, then files `f1.c` and `f2.c` also claim to describe that address.

With the `--dangling-debug-address` option, the address of both functions is set to the specified address. In this example, the address is set to an unused address, which is `0x0000ff` as shown for `image1.axf`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.29 --datacompressor=opt

Enables you to specify one of the supplied algorithms for RW data compression.

Default

The default is `--datacompressor=on`.

Syntax

```
--datacompressor=<opt>
```

Parameters

`<opt>`

One of the following:

`on`

Enables RW data compression to minimize ROM size.

`off`

Disables RW data compression.

`list`

Lists the data compressors available to the linker.

`<id>`

A data compression algorithm:

Table 3-2: Data compressor algorithms

id	Compression algorithm
0	run-length encoding
1	run-length encoding, with LZ77 on small-repeats
2	complex LZ77 compression

Specifying a compressor adds a decompressor to the code area. If the final image does not have compressed data, the decompressor is not added.

Restrictions

Not supported for AArch64 state.

Operation

If you do not specify a data compression algorithm, the linker chooses the most appropriate one for you automatically. In general, it is not necessary to override this choice.

Related information

How compression is applied on page 614

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.30 --debug, --no_debug

Controls the generation of debug information in the output file.

Default

The default is `--debug`.

Syntax

```
--debug
--no_debug
```

Parameters

None.

Operation

Debug information includes debug input sections and the symbol/string table.

Use `--no_debug` to exclude debug information from the output file. The resulting ELF image is smaller, but you cannot debug it at source level. The linker discards any debug input section it finds in the input objects and library members, and does not include the symbol and string table in the image. This only affects the image size as loaded into the debugger. It has no effect on the size of any resulting binary image that is downloaded to the target.

If you are using `--partial` the linker creates a partially-linked object without any debug data.


Note

Do not use the `armlink --no_debug` option if you want to use the `fromelf` options `--expandarrays` and `--fieldoffsets` on the image. The functionality of the `fromelf` options `--expandarrays` and `--fieldoffsets` requires that the object or image file has debug information.

Related information

--fieldoffsets on page 796

3.1.31 --diag_error=tag[,tag,...] (armlink)

Sets diagnostic messages that have a specific tag to Error severity.

Syntax

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

Parameters

`<tag>`

One of:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- 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.

Related information

--diag_remark=tag[,tag,...] (armlink) on page 451
--diag_style=arm|ide|gnu (armlink) on page 451
--diag_suppress=tag[,tag,...] (armlink) on page 452
--diag_warning=tag[,tag,...] (armlink) on page 453
--strict on page 540

3.1.32 --diag_remark=tag[,tag,...] (armlink)

Sets diagnostic messages that have a specific tag to Remark severity.



Note

Remarks are not displayed by default. Use the `--remarks` option to display these messages.

Syntax

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

Parameters

`<tag>`

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 information

--diag_error=tag[,tag,...] (armlink) on page 450
--diag_style=arm|ide|gnu (armlink) on page 451
--diag_suppress=tag[,tag,...] (armlink) on page 452
--diag_warning=tag[,tag,...] (armlink) on page 453
--remarks on page 523
--strict on page 540

3.1.33 --diag_style=arm|ide|gnu (armlink)

Specifies the display style for diagnostic messages.

Default

The default is `--diag_style=arm`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
--diag_style=<string>
```

Parameters

<string>

One of:

arm

Display messages using the legacy 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`.

Operation

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

Related information

--diag_error=tag[,tag,...] (armlink) on page 450
--diag_remark=tag[,tag,...] (armlink) on page 451
--diag_suppress=tag[,tag,...] (armlink) on page 452
--diag_warning=tag[,tag,...] (armlink) on page 453
--remarks on page 523
--strict on page 540

3.1.34 --diag_suppress=tag[,tag,...] (armlink)

Suppresses diagnostic messages that have a specific tag.



Reducing the severity of diagnostic messages might prevent the tool from reporting important faults. Arm recommends that you do not reduce the severity of diagnostics unless you understand the impact on your software.

Syntax

```
--diag_suppress=<tag>[,<tag>,...]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<tag>

One of:

- 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.

Example

To suppress the warning messages that have numbers `L6314W` and `L6305W`, use the following command:

```
armlink --diag_suppress=L6314,L6305 ...
```

Related information

--diag_error=tag[,tag,...] (armlink) on page 450
--diag_remark=tag[,tag,...] (armlink) on page 451
--diag_style=arm|ide|gnu (armlink) on page 451
--diag_warning=tag[,tag,...] (armlink) on page 453
--strict on page 540
--remarks on page 523

3.1.35 --diag_warning=tag[,tag,...] (armlink)

Sets diagnostic messages that have a specific tag to Warning severity.

Syntax

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

Parameters

<tag>

One of:

- 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.

Related information

--diag_error=tag[,tag,...] (armlink) on page 450
--diag_remark=tag[,tag,...] (armlink) on page 451
--diag_style=arm|ide|gnu (armlink) on page 451
--diag_suppress=tag[,tag,...] (armlink) on page 452

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--remarks on page 523

3.1.36 --dynamic_linker=name

Specifies the dynamic linker to use to load and relocate the file at runtime.

Default

The default assumed dynamic linker is `lib/ld-linux.so.3`.

Syntax

```
--dynamic_linker=<name>
--dynamiclinker=<name>
```

Parameters

<name>

`<name>` is the name of the dynamic linker to store in the executable.

Operation

When you link with shared objects, the dynamic linker to use is stored in the executable. This option specifies a particular dynamic linker to use when the file is executed.

This option is only effective when using the *System V* (SysV) linking model with `--sysv`.

Related information

--fini=symbol on page 466
--init=symbol on page 478
--library=name on page 491
SysV Shared Libraries and Executables on page 745

3.1.37 --eager_load_debug, --no_eager_load_debug

Manages how `armlink` loads debug section data.

Default

The default is `--eager_load_debug`.

Syntax

```
--eager_load_debug
--no_eager_load_debug
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

The `--no_eager_load_debug` option causes the linker to remove debug section data from memory after object loading. This lowers the peak memory usage of the linker at the expense of some linker performance, because much of the debug data has to be loaded again when the final image is written.

Using `--no_eager_load_debug` option does not affect the debug data that is written into the ELF file.

Note

If you use some command-line options, such as `--map`, the resulting image or object built without debug information might differ by a small number of bytes. This is because the `.comment` section contains the linker command line used, where the options have differed from the default. Therefore `--no_eager_load_debug` images are a little larger and contain Program Header and possibly a section header a small number of bytes later. Use `--no_comment_section` to eliminate this difference.

Related information

--comment_section, --no_comment_section on page 441

3.1.38 --edit=file_list

Enables you to specify steering files containing commands to edit the symbol tables in the output binary.

Syntax

```
--edit=<file_list>
```

Parameters

`<file_list>`

One of more steering files separated by a comma. Do not include a space after the comma.

Operation

You can specify commands in a steering file to:

- Hide global symbols. Use this option to hide specific global symbols in object files. The hidden symbols are not publicly visible.
- Rename global symbols. Use this option to resolve symbol naming conflicts.

Examples

```
--edit=file1 --edit=file2 --edit=file3

--edit=file1,file2,file3
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Steering file command summary on page 639
Hide and rename global symbols with a steering file on page 641
Linker Steering File Command Reference on page 759

3.1.39 --eh_frame_hdr

When an AArch64 image contains C++ exceptions, merges all `.eh_frame` sections into one `.eh_frame` section and then creates the `.eh_frame_hdr` section.

Default

The default is `--eh_frame_hdr`.

Syntax

```
--eh_frame_hdr
```

Parameters

None.

Restrictions

Valid only for AArch64 images.

Operation

The `.eh_frame_hdr` section contains a binary search table of pointers to the `.eh_frame` records. During the merge `armlink` removes any orphaned records.

Only `.eh_frame` sections defined by the *Linux Standard Base* specification are supported. The `.eh_frame_hdr` section is created according to the *Linux Standard Base* specification. If `armlink` finds an unexpected `.eh_frame` section, it stops merging, does not create the `.eh_frame_hdr` section, and generates corresponding warnings.

Related information

Linux Foundation

3.1.40 --elf-output-format

Modifies the symbols and addresses of the output image to be compatible with third-party tools.

Default

The default is `arm`.

Syntax

```
--elf-output-format=[arm|gnu]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

arm

Modifies an ELF file to comply with the usual Arm® Compiler for Embedded behavior.

gnu

Modifies an ELF file to:

1. Set the `vaddr` of segments to represent the address of the first execution address.
2. Turn off scatter-loading and prevent the region table from being generated in the resulting object. That is, there is no call to the `__scatterload` function.


Note

Setting `--elf-output-format=gnu` implies `--no-scatterload-enabled`.

Restrictions

Restrictions apply to scatter files when using `--elf-output-format=gnu`. Consider the following scatter file fragment:

```
LOAD_REGION <load_address> {
    EXEC_REGION <execution_address> {
        *(+RO)
    }
    ...
}
```

In this scatter file fragment, the following restrictions apply:

- All execution regions within a load region must be placed contiguously. This restriction might require you to split a non-contiguous load region into multiple load regions such that all execution regions within the load region are contiguous.
- The value `<execution_address> - <load_address>` must be a constant value for all execution regions containing non-ZI data.
- All load regions must place ZI data at the end of the load region.


Note

`armlink` cannot report errors or warnings for non-conforming scatter files.


Note

Setting `--elf-output-format=gnu` does not imply `--datacompressor=off`. However, you must also specify `--datacompressor=off`. Disabling the data compressor helps ensure the value `<execution_address> - <load_address>` is a constant value for all execution regions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

If your scatter file does not conform to the restrictions, then you must use scatter-loading or provide your own copy and zero-initialization functions in your startup code. For an example see ../accessing-and-managing-symbols-with-armlink/methods-of-importing-linker-defined-symbols-in-c-and-cpp .

Operation

With the `gnu` option, scatter-loading is not performed because the loading and zero-initializing from the ELF file into memory is expected to be done by an ELF program loader, such as the ELF loader in GDB. ELF program loaders use the information in the ELF program header to load contents from the file and zero-initialize data. When loading and zero-initialization is performed by a program loader your startup code does not need to include functions to copy code and data from load to execution address, or zero-initialize data.

Examples of using --elf-output-format=gnu

The Examples for the armlink --elf-output-format=gnu command-line option section contains examples of the use of the `--elf-output-format=gnu` option.

Related information

--datacompressor=opt on page 448
--diag_suppress=tag[,tag,...] (armlink) on page 452
--scatterload-enabled, --no-scatterload-enabled on page 531
Building images that are compatible with third-party tools

3.1.41 --emit_debug_overlay_relocs

Outputs only relocations of debug sections with respect to overlaid program sections to aid an overlay-aware debugger.

Syntax

```
--emit_debug_overlay_relocs
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Related information

--emit_debug_overlay_section on page 459
--emit_relocs on page 460
--emit_non_debug_relocs on page 459
Manual overlay support
ABI for the Arm Architecture: Support for Debugging Overlaid Programs

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.42 --emit_debug_overlay_section

Emits a special debug overlay section during static linking.

Syntax

```
--emit_debug_overlay_section
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Operation

In a relocatable file, a debug section refers to a location in a program section by way of a relocated location. A reference from a debug section to a location in a program section has the following format:

```
<debug_section_index, debug_section_offset>, <program_section_index,
 program_section_offset>
```

During static linking the pair of `<program>` values is reduced to single value, the execution address. This is ambiguous in the presence of overlaid sections.

To resolve this ambiguity, use this option to output a `.ARM.debug_overlay` section of type `SHT_ARM_DEBUG_OVERLAY = SHT_LOUSER + 4` containing a table of entries as follows:

```
<debug_section_offset>, <debug_section_index>, <program_section_index>
```

Related information

--emit_debug_overlay_relocs on page 458
--emit_relocs on page 460
Automatic overlay support
Manual overlay support
ABI for the Arm Architecture: Support for Debugging Overlaid Programs

3.1.43 --emit_non_debug_relocs

Retains only relocations from non-debug sections in an executable file.

Syntax

```
--emit_non_debug_relocs
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Restrictions

Not supported for AArch64 state.

Related information

--emit_relocs on page 460

3.1.44 --emit_relocs

Retains all relocations in the executable file. This results in larger executable files.

Syntax

```
--emit_relocs
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Operation

This is equivalent to the GNU ld `--emit-relocs` option.

Related information

--emit_debug_overlay_relocs on page 458
--emit_non_debug_relocs on page 459
ABI for the Arm Architecture: Support for Debugging Overlaid Programs

3.1.45 --entry=location

Specifies the unique initial entry point of the image. Although an image can have multiple entry points, only one can be the initial entry point.

Syntax

```
--entry=<location>
```

Parameters

`<location>`

One of the following:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<entry_address>

A numerical value, for example: `--entry=0x0`

<symbol>

Specifies an image entry point as the address of `<symbol>`, for example: `--entry=reset_handler`

<offset>+<object>(<section>)

Specifies an image entry point as an `<offset>` inside a `<section>` within a particular `<object>`, for example:

```
--entry=8+startup.o(startupseg)
```

There must be no spaces in the argument to `--entry`. The input section and object names are matched without case-sensitivity. You can use the following simplified notation:

- `<object>(<section>)`, if `offset` is zero.
- `<object>`, if there is only one input section. `armlink` generates an error message if there is more than one code input section in `<object>`.


Note

If the entry address of your image is in T32 state, then the least significant bit of the address must be set to 1. If you specify a symbol, the linker does this automatically. For example, if the entry code starts at address `0x8000` in T32 state you must use `--entry=0x8001`.


Note

On Unix systems your shell typically requires the parentheses to be escaped with backslashes. Alternatively, enclose the complete section specifier in double quotes, for example:

```
--entry="8+startup.o(startupseg)"
```

Operation

The image can contain multiple entry points. Multiple entry points might be specified in C or C++ code, or assembler source files with either `.global <name>` for GNU syntax assembly or the `ENTRY` directive for `armasm` legacy assembly. If the name of the initial entry point is not called `main`, then you must use the `--entry` option to identify that initial entry point.

It is also possible that no input objects define an entry point. In such cases, a unique initial entry point must be specified for an image, otherwise the error `L6305E` is generated. The initial entry point specified with the `--entry` option is stored in the executable file header for use by the loader. There can be only one occurrence of this option on the command line. A debugger typically uses this entry address to initialize the Program Counter (PC) when an image is loaded. The initial entry point must meet the following conditions:

- The image entry point must lie within an execution region.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The execution region must be non-overlay, and must be a root execution region, that is `load address == execution address`.


Note

To prevent the sections for any additional entry points from being removed by unused section elimination, you must use the `armlink` option `--keep`.

Example: Build an image from a C program

Create the C program file `main.c` containing `main()`:

```
// main.c
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

Compile with:

```
armclang --target=arm-arm-none-eabi -march=armv8a -c -o main.o main.c
```

The default entry point for C and C++ is `main()`. Therefore, you can link with:

```
armlink main.o -o image.axf
```

Example: Build an image from a GNU syntax assembler file

Create the GNU syntax assembler file `iterate_loop.s` containing the `start` label to identify the entry point:

```
// Iterate round a loop 10 times, adding 1 to a register each time.

    .file    "iterate_loop.s"
    .section .text.start,"ax",%progbits
    .global  start
    .p2align 2
    .type    start,%function
    .code    32
start:
    MOV      r5,#0x64       // R5 = 100
    MOV      r4,#0          // R4 = 0
    B        test_loop      // branch to test_loop
loop:
    ADD      r5,r5,#1       // Add 1 to R5
    ADD      r4,r4,#1       // Add 1 to R4
test_loop:
    CMP      r4,#0xa        // if R4 < 10, branch back to loop
    BLT      loop
    .end
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Compile with:

```
armclang --target=arm-arm-none-eabi -march=armv8a -c -o iterate_loop.o
 iterate_loop.s
```

The entry point is not `main`. Therefore, you must specify the initial entry point with `--entry` linker command-line option:

```
armlink --entry=start iterate_loop.o -o image.axf
```

Related information

--keep=section_id (armlink) on page 482
--startup=symbol, --no_startup on page 539
-e on page 48
ENTRY directive on page 991
Image entry points

3.1.46 --errors=filename

Redirects the diagnostics from the standard error stream to a specified file.

Syntax

```
--errors=<filename>
```

Parameters

`<filename>`

The file to contain the diagnostics from the standard error stream.

Operation

The specified file is created at the start of the link stage. If a file of the same name already exists, it is overwritten.

If `<filename>` is specified without path information, the file is created in the current directory.


Note

`--errors=<filename>` does not redirect any output sent to `stderr` as a result of specifying `--summary-stderr`.

Related information

--diag_error=tag[,tag,...] (armlink) on page 450
--diag_remark=tag[,tag,...] (armlink) on page 451
--diag_style=arm|ide|gnu (armlink) on page 451

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--diag_suppress=tag[,tag,...] (armlink) on page 452
--diag_warning=tag[,tag,...] (armlink) on page 453
--remarks on page 523
--summary_stderr, --no_summary_stderr on page 545

3.1.47 --exceptions, --no_exceptions

Controls the generation of exception tables in the final image.

Default

The default is `--exceptions`.

Syntax

```
--exceptions
--no_exceptions
```

Parameters

None.

Operation

Using `--no_exceptions` generates an error message if any exceptions sections are present in the image after unused sections have been eliminated. Use this option to ensure that your code is exceptions free.

3.1.48 --export_all, --no_export_all

Controls the export of all global, non-hidden symbols to the dynamic symbols table.

Default

The default is `--export_all` for building shared libraries and dynamically linked libraries (DLLs).

The default is `--no_export_all` for building applications.

Syntax

```
--export_all
--no_export_all
```

Parameters

None.

Operation

Use `--export_all` to dynamically export all global, non-hidden symbols from the executable or DLL to the dynamic symbol table. Use `--no_export_all` to prevent the exporting of symbols to the dynamic symbol table.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`--export_all` always exports non-hidden symbols into the dynamic symbol table. The dynamic symbol table is created if necessary.

You cannot use `--export_all` to produce a statically linked image because it always exports non-hidden symbols, forcing the creation of a dynamic segment.

For more precise control over the exporting of symbols, use one or more steering files.

Related information

--export_dynamic, --no_export_dynamic on page 465

3.1.49 --export_dynamic, --no_export_dynamic

Controls the export of dynamic symbols to the dynamic symbols table.

Default

`--no_export_dynamic` is the default.

Syntax

```
--export_dynamic
--no_export_dynamic
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Operation

If an executable has dynamic symbols, then `--export_dynamic` exports all externally visible symbols.

`--export_dynamic` exports non-hidden symbols into the dynamic symbol table only if a dynamic symbol table already exists.

You can use `--export_dynamic` to produce a statically linked image if there are no imports or exports.

Related information

--export_all, --no_export_all on page 464

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.50 --filtercomment, --no_filtercomment

Controls whether or not the linker modifies the `.comment` section to assist merging.

Default

The default is `--filtercomment`.

Syntax

```
--filtercomment
--no_filtercomment
```

Parameters

None.

Restrictions

`armlink` does not preprocess comment sections from `armclang`.

Operation

The linker always removes identical comments. The `--filtercomment` permits the linker to preprocess the `.comment` section and remove some information that prevents merging.

Use `--no_filtercomment` to prevent the linker from modifying the `.comment` section.

Related information

Linker merging of comment sections on page 619
--comment_section, --no_comment_section on page 441

3.1.51 --fini=symbol

Specifies the symbol name to use to define the entry point for finalization code.

Syntax

```
--fini=<symbol>
```

Parameters

`<symbol>`

The symbol name to use for the entry point to the finalization code.

Operation

The dynamic linker executes this code when it unloads the executable file or shared object.

Related information

--dynamic_linker=name on page 454
--init=symbol on page 478

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--library=name on page 491

3.1.52 --first=section_id

Places the selected input section first in its execution region. This can, for example, place the section containing the vector table first in the image.

Syntax

```
--first=<section_id>
```

Parameters

<section_id>

One of the following:

<symbol>

Selects the section that defines `<symbol>`. For example: `--first=reset`.

You must not specify a symbol that has more than one definition, because only one section can be placed first.

<object>(<section>)

Selects `<section>` from `<object>`. There must be no space between `<object>` and the following open parenthesis. For example: `--first=init.o(init)`.

<object>

Selects the single input section in `<object>`. For example: `--first=init.o`.

If you use this short form and there is more than one input section in `<object>`, `armlink` generates an error message.

Note

On Unix systems your shell typically requires the parentheses to be escaped with backslashes. Alternatively, enclose the complete section specifier in double quotes, for example:

```
--first="init.o(init)"
```

Operation

The `--first` option cannot be used with `--scatter`. Instead, use the `+FIRST` attribute in a scatter file.

Related information

--last=section_id on page 489
--scatter=filename on page 529
Section placement with the FIRST and LAST attributes on page 594
Section placement with the linker on page 590

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.53 --force_explicit_attr

Causes the linker to retry the CPU mapping using build attributes constructed when an architecture is specified with `--cpu`.


Note

Build attribute compatibility checking is supported only for AArch32 state.

Syntax

```
--force_explicit_attr
```

Parameters

None.

Operation

The `--cpu` option checks the FPU attributes if the CPU chosen has a built-in FPU.

The error message `L6463U : Input Objects contain <archtype> instructions but could not find valid target for <archtype> architecture based on object attributes. Suggest using --cpu option to select a specific cpu.` is given in the following situations:

- The ELF file contains instructions from architecture `<archtype>` yet the build attributes claim that `<archtype>` is not supported.
- The build attributes are inconsistent enough that the linker cannot map them to an existing CPU.

If setting the `--cpu` option still fails, use `--force_explicit_attr` to cause the linker to retry the CPU mapping using build attributes constructed from `--cpu=<archtype>`. This might help if the error is being given solely because of inconsistent build attributes.

Related information

--cpu=name (armlink) on page 443
--fpu=name (armlink) on page 470

3.1.54 --force_so_throw, --no_force_so_throw

Controls the assumption made by the linker that an input shared object might throw an exception.

Default

The default is `--no_force_so_throw`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
--force_so_throw
--no_force_so_throw
```

Parameters

None.

Operation

By default, exception tables are discarded if no code throws an exception.

Use `--force_so_throw` to specify that all shared objects might throw an exception and so force the linker to keep the exception tables, regardless of whether the image can throw an exception or not.

3.1.55 --fpic

Enables you to link *Position Independent Code* (PIC). That is, code that is compiled using the `-fbare-metal-pie` or `-fpic` compiler command-line options.

The `--fpic` option is implicitly specified when the `--bare_metal_pie` option is used.

Syntax

```
--fpic
```

Parameters

None.

Related information

--shared on page 532
--sysv on page 548
Bare-metal Position Independent Executables

3.1.56 --fpu=list (armlink)

Lists the *Floating Point Unit* (FPU) architectures that are supported by the `--fpu=name` option.

Deprecated options are not listed.

Syntax

```
--fpu=list
```

Parameters

`list`

List of supported FPU architectures.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--cpu=list (armlink) on page 442
--cpu=name (armlink) on page 443
--fpu=name (armlink) on page 470

3.1.57 --fpu=name (armlink)

Specifies the target FPU architecture.

Default

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

If the processor you specify with `--cpu` has a VFP coprocessor, the default target FPU architecture is the VFP architecture for that processor.

Syntax

```
--fpu=<name>
```

Parameters

`<name>`

The name of the target FPU architecture. Specify `--fpu=list` to list the supported FPU architecture names that you can use with `--fpu=<name>`.

The default floating-point architecture depends on the target architecture.

Restrictions

Software floating-point linkage is available only for AArch32 state and for Arm®v8-R AArch64 targets.

For Armv8-R AArch64, if you do not link with `--fpu=SoftVFP`, the linker does not report an error.

Arm® Neon® support is disabled for `SoftVFP`.

Operation

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 linker uses this option to optimize the choice of system libraries. The default is to select an FPU that is compatible with all of the component object files.

The linker fails if any of the component object files rely on features that are incompatible with the selected FPU architecture.

Related information

--cpu=list (armlink) on page 442

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--cpu=name (armlink) on page 443
--fpu=list (armlink) on page 469

3.1.58 --got=type

Generates *Global Offset Tables* (GOTs) to resolve GOT relocations in bare metal images. `armlink` statically resolves the GOT relocations.

Default

The default for AArch32 state is `none`.

The default for AArch64 state is `local`.

Syntax

```
--got=<type>
```

Parameters

`<type>`

One of the following:

`none`

Disables GOT generation.

`local`

Creates a local offset table for each execution region.

`global`

Creates a single offset table for the whole image.

Restrictions

The `local` option is not supported for AArch32 state.

3.1.59 --gnu_linker_defined_syms

Enables support for the GNU equivalent of input section symbols.



Note

The `--gnu_linker_defined_syms` linker option is deprecated.

Syntax

```
--gnu_linker_defined_syms
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

If you want GNU-style behavior when treating the Arm® symbols `SectionName$$Base` and `SectionName$$Limit`, then specify `--gnu_linker_defined_syms`.

Table 3-3: GNU equivalent of input sections

GNU symbol	Arm symbol	Description
`__start_<SectionName>`	`<SectionName>$$Base`	Address of the start of the consolidated section called `<SectionName>`.
`__stop_<SectionName>`	`<SectionName>$$Limit`	Address of the byte beyond the end of the consolidated section called `<SectionName>`.

Note

- A reference to `<SectionName>` by a GNU input section symbol is sufficient for `armlink` to prevent the section from being removed as unused.
- A reference by an Arm input section symbol is not sufficient to prevent the section from being removed as unused.

3.1.60 --help (armlink)

Displays a summary of the main command-line options.

Default

This option is the default if you specify the tool command without any options or source files.

Syntax

```
--help
```

Parameters

None.

Related information

--version_number (armlink) on page 558
--vsn (armlink) on page 559

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.61 --import_cmse_lib_in=filename

Reads an existing import library and creates gateway veneers with the same address as given in the import library. This option is useful when producing a new version of a Secure image where the addresses in the output import library must not change. It is optional for a Secure image.

Syntax

```
--import_cmse_lib_in=<filename>
```

Parameters

<filename>

The name of the import library file.

Operation

The input import library is an object file that contains only a symbol table. Each symbol specifies an absolute address of a secure gateway veneer for an entry function of the same name as the symbol.

`armlink` places secure gateway veneers generated from an existing import library using the `__at` feature. New secure gateway veneers must be placed using a scatter file.

Related information

Generation of secure gateway veneers on page 601
--import_cmse_lib_out=filename on page 473
Overview of building Secure and Non-secure images with the Armv8-M Security Extension

3.1.62 --import_cmse_lib_out=filename

Outputs the secure code import library to the location specified. This option is required for a Secure image.

Syntax

```
--import_cmse_lib_out=<filename>
```

Parameters

<filename>

The name of the import library file.

The output import library is an object file that contains only a symbol table. Each symbol specifies an absolute address of a secure gateway for an entry function of the same name as the symbol. Secure gateways include both secure gateway veneers generated by `armlink` and any other secure gateways for entry functions found in the image.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Generation of secure gateway veneers on page 601
--import_cmse_lib_in=filename on page 472
Overview of building Secure and Non-secure images with the Armv8-M Security Extension

3.1.63 --import_unresolved, --no_import_unresolved

Enables or disables the importing of unresolved references when linking SysV shared objects.

Default

The default is `--import_unresolved`.

Syntax

```
--import_unresolved
--no_import_unresolved
```

Parameters

None.

Operation

When linking a shared object with `--sysv --shared` unresolved symbols are normally imported.

If you explicitly list object files on the linker command-line, specify the `--no_import_unresolved` option so that any unresolved references cause an undefined symbol error rather than being imported.

This option is only effective when using the System V (`--sysv`) linking model and building a shared object (`--shared`).

Related information

--shared on page 532
--sysv on page 548

3.1.64 --info=topic[,topic,...] (armlink)

Prints information about specific topics. You can write the output to a text file using `--list=<file>`.

Syntax

```
--info={<topic>[,<topic>,...]}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<topic>

A comma-separated list from the following topic keywords:

any

For unassigned sections that are placed using the `.ANY` module selector, lists:

- The sort order.
- The placement algorithm.
- The sections that are assigned to each execution region in the order that the placement algorithm assigns them.
- Information about the contingency space and policy that is used for each region.

This keyword also displays additional information when you use the execution region attribute `ANY_SIZE` in a scatter file.

architecture

Summarizes the image architecture by listing the processor, FPU, and byte order.

auto_overlay

Writes out a text summary of each overlay that is output.

bti

Outputs a list of BTI and non-BTI user objects. This option is useful when linking BTI with non-BTI user objects.

common

Lists all common sections that are eliminated from the image. Using this option implies `--info=common,totals`.

compression

Gives extra information about the RW compression process.

debug

Lists all rejected input debug sections that are eliminated from the image as a result of using `--remove`. Using this option implies `--info=debug,totals`.

exceptions

Gives information on exception table generation and optimization.

inline

If you also specify `--inline`, lists all functions that the linker inlines, and the total number inlined.

inputs

Lists the input symbols, objects, and libraries.

libraries

Lists the full path name of every library the link stage automatically selects.

You can use this option with `--info_lib_prefix` to display information about a specific library.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

merge

Lists the `const` strings that the linker merges. Each item lists the merged result, the strings being merged, and the associated object files.

pac

Outputs a list of PAC objects and non-PAC objects. This option is useful when linking PAC objects with non-PAC objects.

pltgot

Lists the PLT entries that are built for the executable or DLL.

sizes

Lists the code and data (RO Data, RW Data, ZI Data, and Debug Data) sizes for each input object and library member in the image. Using this option implies `--info=sizes,totals`.

stack

Lists the stack usage of all functions. For AArch64 state this option requires debug information to have been generated by the compiler using the `-g` option for `armclang`.

summarysizes

Summarizes the code and data sizes of the image. Use the `armlink` option `-summary_stderr` to redirect this summary to `stderr`.

summarystack

Summarizes the stack usage of all global symbols. For AArch64 state this option requires debug information to have been generated by the compiler using the `-g` option for `armclang`.

tailreorder

Lists all the tail calling sections that are moved above their targets, as a result of using `--tailreorder`.

totals

Lists the totals of the code and data (RO Data, RW Data, ZI Data, and Debug Data) sizes for input objects and libraries.

unused

Lists all unused sections that are eliminated from the user code as a result of using `--remove`. It does not list any unused sections that are loaded from the Arm C libraries.

unusedsymbols

Lists all symbols that unused section elimination removes.

veneers

Lists the linker-generated veneers.

veneercallers

Lists the linker-generated veneers with additional information about the callers to each veneer. Use with `--verbose` to list each call individually.

veneerpools

Displays information on how the linker has placed veneer pools.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

visibility

Lists the symbol visibility information. You can use this option with either `--info=inputs` or `--verbose` to enhance the output.

weakrefs

Lists all symbols that are the target of weak references, and whether they were defined.

Restrictions

We do not recommend using the `--info=stack` or `--info=summarystack` options with the `armclang` option `-fno-omit-frame-pointer` to measure the stack usage of the function. Local variables on the stack that are accessed through the frame pointer are not counted in the size of the stack frame. Therefore, `armlink` underestimates the amount of stack used.

Operation

The output from `--info=sizes,totals` always includes the padding values in the totals for input objects and libraries.

If you are using RW data compression (the default), or if you have specified a compressor using the `--datacompressor=<id>` option, the output from `--info=sizes,totals` includes an entry under `Grand Totals` to reflect the true size of the image.



Note

Spaces are not permitted between topic keywords in the list. For example, you can enter `--info=sizes,totals` but not `--info=sizes, totals`.



Note

RW data compression is not supported in AArch64.

Related information

--any_contingency on page 425
--any_sort_order=order on page 427
--callgraph, --no_callgraph on page 434
--info_lib_prefix=opt on page 478
--merge, --no_merge on page 507
--veneer_inject_type=type on page 555
Elimination of unused sections on page 611
Manual placement of unassigned sections on page 670
--datacompressor=opt on page 448
--inline, --no_inline on page 479
--remove, --no_remove on page 523
--require-bti on page 524

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--check_pac_mismatch on page 440
--keep_intermediate on page 487
--tailreorder, --no_tailreorder on page 549
--summary_stderr, --no_summary_stderr on page 545
Considerations when working with RW data compression on page 615
Optimization with RW data compression on page 612
How the linker chooses a compressor on page 613
How compression is applied on page 614
Execution region attributes on page 723
-fomit-frame-pointer, -fno-omit-frame-pointer on page 77
Getting Image Details

3.1.65 --info_lib_prefix=opt

Specifies a filter for the `--info=libraries` option. The linker only displays the libraries that have the same prefix as the filter.

Syntax

```
--info=libraries --info_lib_prefix=<opt>
```

Parameters

`<opt>`

The prefix of the required library.

Examples

Displaying a list of libraries without the filter produces a list of libraries, for example:

```
armlink --info=libraries test.o

install_directory\lib\armlib\c_4.l
install_directory\lib\armlib\fz_4s.l
install_directory\lib\armlib\h_4.l
install_directory\lib\armlib\m_4s.l
install_directory\lib\armlib\vfpsupport.l
```

Displaying a list of libraries with the filter produces a list of libraries with the specified prefix, for example:

```
armlink --info=libraries --info_lib_prefix=c test.o

install_directory\lib\armlib\c_4.l
```

Related information

--info=topic[,topic,...] (armlink) on page 474

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.66 --init=symbol

Specifies a symbol name to use for the initialization code. A dynamic linker executes this code when it loads the executable file or shared object.

Syntax

```
--init=<symbol>
```

Parameters

<symbol>

The symbol name you want to use to define the location of the initialization code.

Related information

--dynamic_linker=name on page 454
--fini=symbol on page 466
--library=name on page 491

3.1.67 --inline, --no_inline

Enables or disables branch inlining to optimize small function calls in your image.

Default

The default is `--no_inline`.



This branch optimization is off by default because enabling it changes the image such that debug information might be incorrect. If enabled, the linker makes no attempt to correct the debug information.

`--no_inline` turns off inlining for user-supplied objects only. The linker still inlines functions from the Arm standard libraries by default.

Syntax

```
--inline
--no_inline
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Related information

--branchnop, --no_branchnop on page 434

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Function inlining with the linker on page 615
--inline_type=type on page 480
--tailreorder, --no_tailreorder on page 549

3.1.68 --inline_type=type

Inlines functions from all objects, Arm standard libraries only, or turns off inlining completely.

Syntax

```
--inline_type=<type>
```

Parameters

<type>

One of:

all

The linker is permitted to inline functions from all input objects.

library

The linker is permitted to inline functions from the Arm standard libraries.

none

The linker is not permitted to inline functions.

Operation

This option takes precedence over `--inline` if both options are present on the command line. The mapping between the options is:

- `--inline` maps to `--inline_type=all`
- `--no_inline` maps to `--inline_type=library`



Note

To disable linker inlining completely you must use `--inline_type=none`.

Related information

--inline, --no_inline on page 479
--tailreorder, --no_tailreorder on page 549

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.69 --inlineveneer, --no_inlineveneer

Enables or disables the generation of inline veneers to give greater control over how the linker places sections.

Default

The default is `--inlineveneer`.

Syntax

```
--inlineveneer
--no_inlineveneer
```

Parameters

None.

Related information

--piveneer, --no_piveneer on page 516
--veneershare, --no_veneershare on page 557
Veneer types on page 599
Linker-generated veneers on page 597
Veneer sharing on page 598
Generation of position independent to absolute veneers on page 600
Reuse of veneers when scatter-loading on page 600

3.1.70 input-file-list (armlink)

A space-separated list of objects, libraries, or symbol definitions (symdefs) files.

Operation

The linker sorts through the input file list in order. If the linker is unable to resolve input file problems then a diagnostic message is produced.

The symdefs files can be included in the list to provide global symbol addresses for previously generated image files.

You can use libraries in the input file list in the following ways:

- Specify a library to be added to the list of libraries that the linker uses to extract members if they resolve any non weak unresolved references. For example, specify `mystring.lib` in the input file list.



Note

Members from the libraries in this list are added to the image only when they resolve an unresolved non weak reference.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Specify particular members to be extracted from a library and added to the image as individual objects. Members are selected from a comma separated list of patterns that can include wild characters. Spaces are permitted but if you use them you must enclose the whole input file list in quotes.

  The following shows an example of an input file list both with and without spaces:

  ```
  mystring.lib(strcmp.o,std*.o)

  "mystring.lib(strcmp.o, std*.o)"
  ```

The linker automatically searches the appropriate C and C++ libraries to select the best standard functions for your image. You can use `--no_scanlib` to prevent automatic searching of the standard system libraries.

The linker processes the input file list in the following order:

1. Objects are added to the image unconditionally.
2. Members selected from libraries using patterns are added to the image unconditionally, as if they are objects. For example, to add all `a*.o` objects and `stdio.o` from `mystring.lib` use the following:

   ```
   "mystring.lib(stdio.o, a*.o)"
   ```

3. Library files listed on the command-line are searched for any unresolved non-weak references. The standard C or C++ libraries are added to the list of libraries that the linker later uses to resolve any remaining references.

Related information

--scanlib, --no_scanlib on page 529
Access symbols in another image on page 635
How the linker performs library searching, selection, and scanning on page 606
--stdlib on page 539

3.1.71 --keep=section_id (armlink)

Specifies input sections that must not be removed by unused section elimination. For example, a section in an object that is to be used as an entry point, but is not the initial entry point.

Syntax

```
--keep=<section_id>
```

Parameters

`<section_id>`

One of the following:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<symbol>

Specifies that an input section defining <symbol> is to be retained during unused section elimination. If multiple definitions of <symbol> exist, armlink generates an error message.

For example, you might use --keep=int_handler.

To keep all sections that define a symbol ending in _handler, use --keep=*_handler.

<object>(<section>)

Specifies that <section> from <object> is to be retained during unused section elimination. If a single instance of <section> is generated, you can omit <section>, for example, file.o(). Otherwise, you must specify <section>.

For example, to keep the vect section from the vectors.o object use:

```
--keep=vectors.o(vect)
```

To keep all sections from the vectors.o object where the first three characters of the name of the sections are vec, use: --keep=vectors.o(vec*)

<object>

Specifies that the single input section from <object> is to be retained during unused section elimination. If you use this short form and there is more than one input section in <object>, the linker generates an error message.

For example, you might use --keep=dspdata.o.

To keep the single input section from each of the objects that has a name starting with dsp, use --keep=dsp*.o.

Note

On Unix systems your shell typically requires that you escape parentheses with backslashes. Alternatively, enclose the complete section specifier in double quotes, for example:

```
--keep="foo.o(Premier*)"
```

Operation

All forms of the <section_id> argument can contain the * and ? wild characters. Matching is case-insensitive, even on hosts with case-sensitive file naming. For example:

- --keep foo.o(Premier*) causes the entire match for Premier* to be case-insensitive.
- --keep foo.o(Premier) causes a case-insensitive match for the string Premier.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

The only case where a case-sensitive match is made is for `--keep=<symbol>` when `<symbol>` does not contain any wildcard characters.

Use `*.o` to match all object files. Use `*` to match all object files and libraries.

You can specify multiple `--keep` options on the command line.

Matching a symbol that has the same name as an object

If you name a symbol with the same name as an object, then `--keep=<symbol>` searches for a symbol that matches `<symbol_id>`:

- If a symbol is found, it matches the symbol.
- If no symbol is found, it matches the object.

You can force `--keep` to match an object with `--keep=<symbol_id>()`. Therefore, to keep both the symbol and the object, specify `--keep foo.o --keep foo.o()`.

Example: C program with vector table in assembly

Create the file `main.c` containing:

```
// main.c
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

Create the GNU syntax assembler file `vectors.s` containing a vector table. The file contains sufficient code to show how the `--keep` option works, and is not necessarily a complete example:

```
// vectors.s
  .section StartUp, "ax"
  .balign 0x20
  .global Vectors

Vectors:
  B       Reset_Handler
  B       Undefined_Handler
  B       SVC_Handler
  B       Prefetch_Handler
  B       Abort_Handler
  B .     //Reserved vector
  B       IRQ_Handler
  B       FIQ_Handler
// -----------------------------------------------------------
// Handlers for unused exceptions
// -----------------------------------------------------------

Undefined_Handler:
  B       Undefined_Handler
SVC_Handler:
  B       SVC_Handler
Prefetch_Handler:
  B       Prefetch_Handler
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
Abort_Handler:
  B       Abort_Handler
FIQ_Handler:
  B       FIQ_Handler

  .global main

  .global IRQ_Handler
  .type IRQ_Handler, "function"
IRQ_Handler:
  // IRQ handler code

  .global Reset_Handler
  .type Reset_Handler, "function"
Reset_Handler:
  // Reset_Handler code

  //
  // Branch to application
  // ---------------------
  B       main
```

Compile the C and assembler programs to create the corresponding object files:

```
armclang -c --target=arm-arm-none-eabi -march=armv8a -o main.o main.c
armclang -c --target=arm-arm-none-eabi -march=armv8a -o vectors.o vectors.s
```

Link the object files without `--keep`:

```
armlink vectors.o main.o -o image.axf
```

Link the object files with `--keep=Vectors`:

```
armlink --keep=Vectors vectors.o main.o -o image_keep.axf
```

Disassemble the image files with `fromelf`, and compare the results:

```
fromelf --disassemble image.axf --output=image.s
fromelf --disassemble image_keep.axf --output=image_keep.s
```

The file `image_keep.s` contains the disassembly for `vectors.s`.

Example: Multiple C programs with one unused function

Create the C files `test1.c`, `test2.c`, and `unused.c`:

```
// test1.c
#include <stdio.h>

extern int test2(void);

int main(void)
{
    printf("%d\n", test2());
    return 0;
}

// test2.c
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
int test2(void) { return 1; }

// unused.c
int x(void) { return 0; }
```

Create the scatter file `test.scat`:

```
LR  0x8000
{
    ER_RO   +0
    {
        *(+RO)
    }
    ER_VENEERS 0x400000
    {
        *(Veneer$$Code)
        unused.o(+RO)
    }
    ER_TEST2 0x40000000
    {
        test2.o(+RO)
    }
    ER_RWZI 0x40008000
    {
        *(+RW,+ZI)
    }
    ARM_LIB_STACKHEAP +0 EMPTY 0x8000
    {
    }
}
```

Compile and link the files:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a8 -c test1.c
armclang --target=arm-arm-none-eabi -mcpu=cortex-a8 -c test2.c
armclang --target=arm-arm-none-eabi -mcpu=cortex-a8 -c unused.c
armlink -o broken.axf --scatter test.scat test1.o test2.o unused.o
```

`armlink` generates a region table entry in preparation for adding sections at a later stage in its processing. In this example, we have placed `unused.o` in the `ER_VENEERS` execution region. This execution region is also used for linker-generated sections, such as veneers to enable long branches. Therefore, if the linker removes `unused.o` by the unused section elimination feature, then the linker also removes the region table entry associated with this execution region. Subsequently, when the linker later attempts to assign linker-generated sections to the execution region, it cannot, and the following error is output:

```
Error: L6077E: Region table entry for region ER_VENEERS is missing.
"test.scat", line 10 (column 19): Warning: L6329W: Pattern unused.o(RO) only matches
 removed unused sections.
Finished: 0 information, 1 warning and 1 error messages.
```

To fix this error, link with the `--keep` option:

```
armlink -o ok.axf --scatter test.scat test1.o test2.o unused.o --keep unused.o(*)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--entry=location on page 460
How the linker performs library searching, selection, and scanning on page 606
The structure of an Arm ELF image on page 570
Image entry points on page 577
Example startup code for Arm Compiler for Embedded 6 project
The vector table

3.1.72 --keep_intermediate

Specifies whether the linker preserves the ELF intermediate object file produced by the link-time optimizer.

Default

By default, `armlink` does not preserve the intermediate object file produced by the link-time optimizer.

Syntax

```
--keep_intermediate=<option>
```

Parameters

`<option>` is:

`lto`

Preserve an intermediate ELF object file produced by the link-time optimizer.



You cannot use the LTO intermediate file as input to other link commands. The reason is that the build attributes on the intermediate file might not match those of other object files in a link. This situation can cause `armlink` to select the wrong library variants or generate an error if you specify `--cpu`.

Related information

--lto, --no_lto on page 498
Optimizing across modules with Link-Time Optimization

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.73 --largeregions, --no_largeregions

Controls the sorting order of sections in large execution regions to minimize the distance between sections that call each other.

Default

The default is `--no_largeregions`. The linker automatically switches to `--largeregions` if at least one execution region contains more code than the smallest inter-section branch. The smallest inter-section branch depends on the code in the region and the target processor:

128MB
Execution region contains only A64 instructions.

32MB
Execution region contains only A32 instructions.

16MB
Execution region contains T32 instructions, 32-bit T32 instructions are supported.

4MB
Execution region contains T32 instructions, no 32-bit T32 instructions are supported.

Syntax

```
--largeregions
--no_largeregions
```

Parameters

None.

Operation

If the execution region contains more code than the range of a branch instruction then the linker switches to large region mode. In this mode the linker sorts according to the approximated average call depth of each section in ascending order. The linker might also distribute veneers amongst the code sections to minimize the number of veneers.


Note

Large region mode can result in large changes to the layout of an image even when small changes are made to the input.

To disable large region mode and revert to lexical order, use `--no_largeregions`. Section placement is then predictable and image comparisons are more predictable. The linker automatically switches on `--veneerinject` if it is needed for a branch to reach the veneer.

Large region support enables:

- Average call depth sorting, `--sort=AvgCallDepth`.
- API sorting, `--api`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Veneer injection, `--veneerinject`.

The following command lines are equivalent:

```
armlink --largeregions --no_api --no_veneerinject --sort=Lexical
armlink --no_largeregions
```

Related information

--api, --no_api on page 428
--sort=algorithm on page 535
--veneer_inject_type=type on page 555
Linker-generated veneers on page 597
Veneer sharing on page 598
Veneer types on page 599
Generation of position independent to absolute veneers on page 600
--veneerinject, --no_veneerinject on page 554

3.1.74 --last=section_id

Places the selected input section last in its execution region.

Syntax

```
--last=<section_id>
```

Parameters

`<section_id>`

One of the following:

`<symbol>`

Selects the section that defines `<symbol>`. You must not specify a symbol that has more than one definition because only a single section can be placed last. For example, `--last=checksum`.

`<object>(<section>)`

Selects the `<section>` from `<object>`. There must be no space between `<object>` and the following open parenthesis. For example, `--last=checksum.o(check)`.

`<object>`

Selects the single input section from `<object>`. For example, `--last=checksum.o`.

If you use this short form and there is more than one input section in `<object>`, `armlink` generates an error message.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

On Unix systems your shell typically requires the parentheses to be escaped with backslashes. Alternatively, enclose the complete section specifier in double quotes, for example:

```
--last="checksum.o(check)"
```

Operation

The `--last` option cannot be used with `--scatter`. Instead, use the `+LAST` attribute in a scatter file.

Example

This option can force an input section that contains a checksum to be placed last in the RW section.

Related information

--first=section_id on page 467
--scatter=filename on page 529
Section placement with the FIRST and LAST attributes on page 594
Section placement with the linker on page 590

3.1.75 --legacyalign, --no_legacyalign

Controls how padding is inserted into the image.

Note

The `--legacyalign` and `--no_legacyalign` linker options are deprecated.

Default

The default is `--no_legacyalign`.

Syntax

```
--legacyalign
--no_legacyalign
```

Parameters

None.

Operation

Using `--legacyalign`, the linker assumes execution regions and load regions to be four-byte aligned. This option enables the linker to minimize the amount of padding that it inserts into the image.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The `--no_legacyalign` option instructs the linker to insert padding to force natural alignment of execution regions. Natural alignment is the highest known alignment for that region.

Use `--no_legacyalign` to ensure strict conformance with the ELF specification.

You can also use expression evaluation in a scatter file to avoid padding.

Related information

Section placement with the linker on page 590
Load region attributes on page 715
Execution region attributes on page 723
Example of using expression evaluation in a scatter file to avoid padding on page 697

3.1.76 --libpath=pathlist

Specifies a list of paths that the linker uses to search for the Arm standard C and C++ libraries.

Syntax

```
--libpath=<pathlist>
```

Parameters

`<pathlist>`

A comma-separated list of paths that the linker only uses to search for required Arm libraries. Do not include spaces between the comma and the path name when specifying multiple path names, for example, `<path1>,<path2>,<path3>,...,<pathn>`.



This option does not affect searches for user libraries. Use `--userlibpath` instead for user libraries.

Related information

--userlibpath=pathlist on page 554

3.1.77 --library=name

Enables the linker to search a static library without you having specifying the full library filename on the command-line.



Not supported in the Keil® Microcontroller Development Kit (Keil® MDK).

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
--library=<name>
```

Parameters

<name>

Links with the static library, `lib<name>.a`.

Operation

The order that references are resolved to libraries is the order that you specify the libraries on the command line.

Example

The following example shows how to search for `libfoo.a` before `libbar.a`:

```
--library=foo --library=bar
```

Related information

--fpic on page 469
--shared on page 532

3.1.78 --library_security=protection

Selects one of the security hardened libraries with varying levels of protection, which include branch protection and memory tagging.



Note

This topic includes descriptions of [ALPHA] features. See Support level definitions.

Default

The default is `--library_security=auto`.

Syntax

```
--library_security=<protection>
```

Parameters

<protection>

Specifies the level of protection in the library:

v8.3a

Selects the `v8.3a` library, which provides branch protection using *Branch Target Identification* (BTI) and pointer authentication on function returns.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`v8.5a` [ALPHA]

Selects the `v8.5a` library, which provides memory tagging protection of the stack and heap used by the library code. This library also includes all the protection in the `v8.3a` library. Use of the `v8.5a` library is an [ALPHA] feature.

`none`

Selects the standard C library that does not provide protection using BTI and pointer authentication, and does not provide memory tagging protection. This level also forces the linker to select a non-PACBTI library and suppresses errors about mixing BTI and non-BTI user objects.

For example, where the linker normally selects `c_xua.l`, specifying `--library_security=none` makes the linker select `c_xu.l`

`pacbti-m`

Forces the linker to always select a Arm®v8.1-M PACBTI library and suppresses errors about mixing BTI and non-BTI user objects.

`auto`

The linker automatically selects either the standard C library, or the `v8.3a`, or the `v8.5a` library. If at least one input object file has been compiled with `-fsanitize=memtag-heap` or `-fsanitize=memtag-stack` and at least one input object file has return address signing with pointer authentication, then the linker selects the `v8.5a` library. Otherwise, if at least one input object file has been compiled for Armv8.3-A or later, and has return address signing with pointer authentication, then the linker selects the `v8.3a` library. Otherwise, the behavior is the same as `--library_security=none`.


Note

- The presence of `BTI` instructions in the compiled objects does not affect automatic library selection.
- The presence of memory tagging instructions in the compiled objects does not affect automatic library selection.

Operation

Use `--library_security` to override the automatic selection of protected libraries for branch protection and memory tagging protection (stack memory tagging and heap memory tagging).

Branch protection increases the protection from *Return Oriented Programming* (ROP) and *Jump Oriented Programming* (JOP) attacks. Branch protection using pointer authentication and BTI are only available in AArch64 state.

Memory tagging stack protection protects accesses to variables on the stack whose addresses are taken. Heap memory tagging protects against heap overflow attacks. Memory tagging protection is available for the AArch64 state for architectures with the memory tagging extension.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

- Selecting the `v8.5a` library does not automatically imply memory tagging protection of the heap (heap memory tagging). To enable heap memory tagging protection, use the `armclang` option `-fsanitize=memtag-heap` and define the `armclang` symbol `__use_memtag_heap`. You can define this symbol irrespective of the level of `<protection>` you use for `--library_security=<protection>`. For more information, see Choosing a heap implementation for memory allocation functions.
- Code that is compiled with memory tagging can be safely linked together with code that is compiled without memory tagging. However, if any object file is compiled with `-fsanitize=memtag-heap` or `-fsanitize=memtag-stack`, and if `setjmp`, `longjmp`, or C++ exceptions are present anywhere in the image, then you must use the `v8.5a` library to avoid tagging related memory fault at runtime.

The linker library selection for PACBTI works as follows:

- If any input object file has the `Tag_PACRET_use` build attribute, that is interpreted as an intent to use PAC-RET and to link to PACBTI libraries. The linker allows mixing PAC-RET with non-PAC-RET objects.
- If any input object file has the `Tag_BTI_use` build attribute, that is interpreted as an intent to use BTI and to link to PACBTI libraries. The linker disallows mixing BTI with non-BTI objects, for the objects explicitly specified on the command-line or from user libraries.
- If none of the input objects have either `Tag_PACRET_use` or `Tag_BTI_use`, the linker does not link to the PACBTI libraries.

Example: v8.3a library with branch protection using BTI and pointer authentication

```
armlink --cpu=8.3-A.64 --library_security=v8.3a foo.o
```

Example: Standard C library without any branch protection using BTI and pointer authentication

```
armlink --cpu=8.3-A.64 --library_security=none foo.o
```

Example: v8.5a library with memory tagging stack protection, and branch protection using BTI and pointer authentication

```
armlink --library_security=v8.5a foo.o
```

Related information

--library_type=lib on page 494
--require-bti on page 524
--check_pac_mismatch on page 440
-mbranch-protection on page 144
__attribute__((target("options"))) function attribute on page 276

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.79 --library_type=lib

Selects the library to be used at link time. Use this option when use of the libraries require more specialized optimizations.

Default

If you do not specify `--library_type` at link time and no object file specifies a preference, then the linker assumes `--library_type=standardlib`.

Syntax

```
--library_type=<lib>
```

Parameters

`<lib>`

One of the following:

`standardlib`

Specifies that the full Arm® Compiler for Embedded runtime libraries are selected at link time.

`microlib`

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

Restrictions

Microlib is not supported for AArch64 state.

Related information

Building an application with microlib

3.1.80 --list=filename

Redirects diagnostic output to a file.

Syntax

```
--list=<filename>
```

Parameters

`<filename>`

The file to use to save the diagnostic output. `<filename>` can include a path

Operation

Redirects the diagnostics output by the `--info`, `--map`, `--symbols`, `--verbose`, `--xref`, `--xreffrom`, and `--xrefto` options to `<file>`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The specified file is created when diagnostics are output. If a file of the same name already exists, it is overwritten. However, if diagnostics are not output, a file is not created. In this case, the contents of any existing file with the same name remain unchanged.

If `<filename>` is specified without a path, it is created in the output directory, that is, the directory where the output image is being written.

Related information

--map, --no_map on page 504
--verbose on page 557
--xref, --no_xref on page 561
--xrefdbg, --no_xrefdbg on page 561
--xreffrom, --xrefto on page 562
--info=topic[,topic,...] (armlink) on page 474
--symbols, --no_symbols on page 546

3.1.81 --list_mapping_symbols, --no_list_mapping_symbols

Enables or disables the addition of mapping symbols in the output produced by `--symbols`.

The mapping symbols `$a`, `$t`, `$t.x`, `$d`, and `$x` flag transitions between A32 code, T32 code, ThumbEE code (Arm®v7-A), data, and A64 code.

Default

The default is `--no_list_mapping_symbols`.

Syntax

```
--list_mapping_symbols
--no_list_mapping_symbols
```

Parameters

None.

Related information

--symbols, --no_symbols on page 546
About mapping symbols on page 622
ELF for the Arm Architecture

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.82 --load_addr_map_info, --no_load_addr_map_info

Includes the load addresses for execution regions and the input sections within them in the map file.

Default

The default is `--no_load_addr_map_info`.

Syntax

```
--load_addr_map_info
--no_load_addr_map_info
```

Parameters

None.

Restrictions

You must use `--map` with this option.

Operation

If an input section is compressed, then the load address has no meaning and `COMPRESSED` is displayed instead.

For sections that do not have a load address, such as ZI data, the load address is blank

Example

The following example shows the format of the map file output:

```
Base Addr    Load Addr    Size         Type   Attr      Idx    E Section Name
 Object
0x00008000   0x00008000   0x00000008   Code   RO           25  * !!!main
  __main.o(c_4.l)
0x00010000   COMPRESSED   0x00001000   Data   RW            2    dataA
 data.o
0x00003000        -       0x00000004   Zero   RW            2    .bss
 test.o
```

Related information

--map, --no_map on page 504

3.1.83 --locals, --no_locals

Adds local symbols or removes local symbols depending on whether an image or partial object is being output.

Default

The default is `--locals`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
--locals
--no_locals
```

Parameters

None.

Operation

The `--locals` option adds local symbols in the output symbol table.

The effect of the `--no_locals` option is different for images and object files.

When producing an executable image `--no_locals` removes local symbols from the output symbol table.

For object files built with the `--partial` option, the `--no_locals` option:

- Keeps mapping symbols and build attributes in the symbol table.
- Removes those local symbols that can be removed without loss of functionality.

  Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.

`--no_locals` is a useful optimization if you want to reduce the size of the output symbol table in the final image.

Related information

--privacy (armlink) on page 520
--privacy (fromelf) on page 816
--strip=option[,option,...] on page 823

3.1.84 --lto, --no_lto

Enables *Link-Time Optimization* (LTO).

Default

The default is `--no_lto`.

Syntax

```
--lto
--no_lto
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

Note

LTO performs aggressive optimizations by analyzing the dependencies between bitcode format objects. Such aggressive optimizations can result in the removal of unused variables and functions in the source code.

Note

When you specify the `-flto` option, `armclang` produces ELF files that contain bitcode in a `.llvm.lto` section.

With the `--no_lto` option, `armlink` gives an error message if it encounters any `.llvm.lto` sections.

Link-time optimization requires the dependent library `libLTO`.

Table 3-4: Link-Time Optimization dependencies

Dependency	Windows filename	Linux filename
`libLTO`	`LTO.dll`	`libLTO.so`

By default, the dependent library `libLTO` is present in the same directory as `armlink`.

The search order for these dependencies is as follows:

`LTO.dll`

1. The same directory as the `armlink` executable.
2. The directories in the current directory search path.

`libLTO.so`

1. The same directory as the `armlink` executable.
2. The directories in the `LD_LIBRARY_PATH` environment variable.
3. The cache file `/etc/ld.so.cache`.
4. The directories `/lib` and `/usr/lib`.

   These directories might have the suffix `64` on some 64-bit Linux systems. For example, on 64-bit Red Hat Enterprise Linux the directories are `/lib64` and `/usr/lib64`.

Note

The `armclang` executable and the `libLTO` library must come from:

- The same Arm® Compiler for Embedded 6 installation.
- The same version of the compiler.

Any use of `libLTO` other than the library supplied with Arm Compiler for Embedded 6 is unsupported.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

LTO does not honor the `armclang` option `-mexecute-only`. If you use the `armclang` options `-flto`, `-Omax`, or `-Omin`, then the compiler cannot generate execute-only code.

Related information

--info=topic[,topic,...] (armlink) on page 474
--keep_intermediate on page 487
--lto_keep_all_symbols, --no_lto_keep_all_symbols on page 500
--lto_intermediate_filename on page 500
--lto_relocation_model on page 503
--lto_level on page 501
-Omax (armlink) on page 512
-flto, -fno-lto on page 72
Optimizing across modules with Link-Time Optimization

3.1.85 --lto_keep_all_symbols, --no_lto_keep_all_symbols

Specifies whether link-time optimization removes unreferenced global symbols.

Default

The default is `--no_lto_keep_all_symbols`.

Syntax

```
--lto_keep_all_symbols
--no_lto_keep_all_symbols
```

Parameters

None.

Operation

Using `--lto_keep_all_symbols` affects all symbols and largely reduces the usefulness of link-time optimization. If you need to keep only a specific unreferenced symbol, then use the `--keep` option instead.

Related information

--keep=section_id (armlink) on page 482
--lto, --no_lto on page 498
Optimizing across modules with Link-Time Optimization

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.86 --lto_intermediate_filename

Specifies the name of the ELF object file produced by the link-time optimizer.

Default

The default is a temporary filename.

Syntax

```
--lto_intermediate_filename=<filename>
```

Parameters

<filename>

The filename the link-time optimizer uses for the ELF object file it produces.

Operation

The purpose of the `--lto_intermediate_filename` option is so that the intermediate file produced by the link-time optimizer can be named in other inputs to the linker, such as scatter-loading files.


Note

The `--lto_intermediate_filename` option does not cause the linker to keep the intermediate object file. Use the `--keep-intermediate=lto` option to keep the intermediate file.


Note

If you keep the LTO intermediate file, you cannot use it as input to other link commands. The reason is that the build attributes on the intermediate file might not match those of other object files in a link. This situation can cause `armlink` to select the wrong library variants or generate an error if you specify `--cpu`.

Related information

--keep_intermediate on page 487
--lto, --no_lto on page 498
Optimizing across modules with Link-Time Optimization

3.1.87 --lto_level

Sets the optimization level for the *Link-Time Optimization* (LTO) feature.

Default

If you do not specify `O<level>`, the linker assumes `O2`.

Syntax

```
--lto_level=O<level>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<level>

One of the following:

0

Minimum optimization for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this optimization might result in a larger image.

1

Restricted optimization. When debugging is enabled, this option selects a good compromise between image size, performance, and quality of debug view.

We recommend `-O1` rather than `-O0` for the best trade-off between debug view, code size, and performance.

2

High optimization. When debugging is enabled, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The linker might perform optimizations that the debug information cannot describe.

This optimization is the default optimization level.

3

Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.

fast

Enables the optimizations from both the `armclang` options `-O3` and `-ffp-mode=fast`.


Note

Enabling the aggressive optimizations that the `armclang` option `-ffp-mode=fast` performs might violate strict compliance with language standards.

max

Maximum optimization. Specifically targets performance optimization. Enables all the optimizations from level `fast`, together with other aggressive optimizations.


Note

This option is not guaranteed to be fully standards-compliant for all code cases.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

- Code-size, build-time, and the debug view can each be adversely affected when using this option.
- We cannot guarantee that the best performance optimization is achieved in all code cases.

s

Performs optimizations to reduce code size, balancing code size against code speed.

z

Performs optimizations to minimize image size.

min

Specifically targets reducing code size. Enables all the optimizations from level `-Oz`, together with LTO aimed at removing unused code and data, and virtual function elimination.


Note

- Performance, build-time, and the debug view can each be adversely affected when using this option.
- We cannot guarantee that the best code size optimization is achieved in all code cases.

Related information

--lto, --no_lto on page 498
-Omax (armlink) on page 512
-O (armclang) on page 203
Optimizing across modules with Link-Time Optimization

3.1.88 --lto_relocation_model

Specifies whether the link-time optimizer produces absolute or position independent code.

Default

The default is `--lto_relocation_model=static`.

Syntax

```
--lto_relocation_model=<model>
```

Parameters

<model>

One of the following:

static

The link-time optimizer produces absolute code.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

pic

The link-time optimizer produces code that uses GOT relative position independent code.

The `--lto_relocation_model=pic` option requires the `armlink` option `--bare_metal_pie`.

Related information

--bare_metal_pie on page 430
--lto, --no_lto on page 498
Optimizing across modules with Link-Time Optimization

3.1.89 --mangled, --unmangled

Instructs the linker to display mangled or unmangled C++ symbol names in diagnostic messages, and in listings produced by the `--xref`, `--xreffrom`, `--xrefto`, and `--symbols` options.

Default

The default is `--unmangled`.

Syntax

```
--mangled
--unmangled
```

Parameters

None.

Operation

If `--unmangled` is selected, C++ symbol names are displayed as they appear in your source code.

If `--mangled` is selected, C++ symbol names are displayed as they appear in the object symbol tables.

Related information

--symbols, --no_symbols on page 546
--xref, --no_xref on page 561
--xrefdbg, --no_xrefdbg on page 561
--xreffrom, --xrefto on page 562

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.90 --map, --no_map

Enables or disables the printing of a memory map.

Default

The default is `--no_map`.

Syntax

```
--map
--no_map
```

Parameters

None.

Operation

The map contains the address and the size of each load region, execution region, and input section in the image, including linker-generated input sections. This can be output to a text file using `--list=<filename>`.

Related information

--load_addr_map_info, --no_load_addr_map_info on page 496
--list=filename on page 495
--section_index_display=type on page 531

3.1.91 --max_er_extension=size

Specifies a constant value to add to the size of an execution region when no maximum size is specified for that region. The value is used only when placing `__at` sections.

Default

The default size is `10240` bytes.

Syntax

```
--max_er_extension=<size>
```

Parameters

`<size>`

The constant value in bytes to use when calculating the size of the execution region.

Related information

Automatic placement of __at sections on page 664

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.92 --max_veneer_passes=value

Specifies a limit to the number of veneer generation passes the linker attempts to make when certain conditions are met.

Default

The default number of passes is 10.

Syntax

```
--max_veneer_passes=<value>
```

Parameters

<value>

The maximum number of veneer passes the linker is to attempt. The minimum value you can specify is 1.

Operation

The linker applies this limit when both the following conditions are met:

- A section that is sufficiently large has a relocation that requires a veneer.
- The linker cannot place the veneer close enough to the call site.

The linker attempts to diagnose the failure if the maximum number of veneer generation passes you specify is exceeded, and displays a warning message. You can downgrade this warning message using `--diag_remark`.

Related information

--diag_remark=tag[,tag,...] (armlink) on page 451
--diag_warning=tag[,tag,...] (armlink) on page 453

3.1.93 --max_visibility=type

Controls the visibility of all symbol definitions.

Default

The default is `--max_visibility=default`.

Syntax

```
--max_visibility=<type>
```

Parameters

<type>

Can be one of:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

default
Default visibility.

protected
Protected visibility.

Operation

Use `--max_visibility=protected` to limit the visibility of all symbol definitions. Global symbol definitions that normally have default visibility, are given protected visibility when this option is specified.

Related information

--override_visibility on page 511

3.1.94 --merge, --no_merge

Enables or disables the merging of `const` strings that the compiler places in shareable sections.

Default

The default is `--merge`.

Syntax

```
--merge
--no_merge
```

Parameters

None.

Operation

If there are similarities between `const` strings, using `--merge` can reduce the size of the image.


Note

With `--merge`, the `OVERLAY` load and execution region attribute prevents merging of `const` strings in those regions with `const` strings in other regions. `const` strings within a region are merged.

Use `--info=merge` to see a listing of the merged const strings.

By default, merging happens between different load and execution regions. Therefore, code from one execution or load region might use a string stored in different region. If you do not want this behavior, then do one of the following:

- Use the `PROTECTED` load region attribute if you are using scatter-loading.
- Globally disable merging with `--no_merge`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--info=topic[,topic,...] (armlink) on page 474
Load region attributes on page 715
Interaction of OVERLAY and PROTECTED attributes with armlink merge options

3.1.95 --merge_litpools, --no_merge_litpools

Attempts to merge identical constants in objects targeted at AArch32 state. The objects must be produced with Arm® Compiler for Embedded 6.

Default

`--merge_litpools` is the default.

Syntax

```
--merge_litpools
--no_merge_litpools
```

Parameters

None.

Operation

The scatter-loading `PROTECTED` and `OVERLAY` attributes modify the behavior of `--merge_litpools`.

`--no_merge_litpools` prevents the merging of literal pool constants, even within the same load region.

Related information

Merging identical constants on page 619
Load region attributes on page 715
Execution region attributes on page 723
Interaction of OVERLAY and PROTECTED attributes with armlink merge options

3.1.96 --muldefweak, --no_muldefweak

Enables or disables multiple weak definitions of a symbol.

Default

The default is `--muldefweak`.

Syntax

```
--muldefweak
--no_muldefweak
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

If enabled, the linker chooses the first definition that it encounters and discards all the other duplicate definitions. If disabled, the linker generates an error message for all multiply defined weak symbols.

3.1.97 -o filename, --output=filename (armlink)

Specifies the name of the output file. The file can be either a partially-linked object or an executable image, depending on the command-line options used.

Syntax

```
--output=<filename>
-o <filename>
```

Parameters

`<filename>`

The name of the output file, and can include a path.

Operation

If `--output=<filename>` is not specified, the linker uses the following default filenames:

`__image.axf`

If the output is an executable image.

`__object.o`

If the output is a partially-linked object.

If `<filename>` is specified without path information, it is created in the current working directory. If path information is specified, then that directory becomes the default output directory.

Related information

--callgraph_file=filename on page 436
--partial on page 515

3.1.98 --output_float_abi=option

Specifies the floating-point procedure call standard to advertise in the ELF header of the executable.

Default

The default option is `auto`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
--output_float_abi=<option>
```

Parameters

<option>

One of the following:

auto

Checks the object files to determine whether the hard float or soft float bit in the ELF header flag is set.

hard

The executable file is built to conform to the hardware floating-point procedure-call standard.

soft

Conforms to the software floating-point procedure-call standard.

Restrictions

Not supported for AArch64 state.

Operation

When the option is set to `auto`:

- For multiple object files:
  - If all the object files specify the same value for the flag, then the executable conforms to the relevant standard.
  - If some files have the hard float and soft float bits in the ELF header flag set to different values from other files, this option is ignored and the hard float and soft float bits in the executable are unspecified.
- If a file has the build attribute `Tag_ABI_VFP_args` set to `2`, then the hard float and soft float bits in the ELF header flag in the executable are set to zero.
- If a file has the build attribute `Tag_ABI_VFP_args` set to `3`, then `armlink` ignores this option.

You can use `fromelf --text` on the image to see whether hard or soft float is set in the ELF header flag.

Related information

--decode_build_attributes on page 784
--text on page 826
ELF for the Arm Architecture
Run-time ABI for the Arm Architecture

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.99 --overlay_veneers

When using the automatic overlay mechanism, this option causes `armlink` to redirect calls between overlays to a veneer. The veneer allows an overlay manager to unload and load the correct overlays.

Syntax

```
--overlay_veneers
```

Parameters

None.

Operation

You must use this option if your scatter file includes execution regions with `AUTO_OVERLAY` attribute assigned to them.


Note

Arm® Compiler for Embedded does not support using both manual and automatic overlays within the same program.

`armlink` creates a veneer for a function call when any of the following are true:

- The calling function is in non-overlaid code and the called function is in an overlay.
- The calling function is in an overlay and the called function is in a different overlay.
- The calling function is in an overlay and the called function is in non-overlaid code.

In the last of these cases, an overlay does not have to be loaded immediately, but the overlay manager typically has to adjust the return address. It does this adjustment so that it can arrange to check on function return that the overlay of the caller is reloaded before returning to it.

Veneers are not created when calls between two functions are in the same overlay. If the calling function is running, then the called function is guaranteed to be loaded already, because each overlay is atomic. This situation is also guaranteed when the called function returns.

A relocation might refer to a function in an overlay and not modify a branch instruction. For example, the relocations R_ARM_ABS32 or R_ARM_REL32 do not modify a branch instruction. In this situation, `armlink` redirects the relocation to point at a veneer for the function regardless of where the relocation is. This redirection is done in case the address of the function is passed into another overlay as an argument.

Related information

Execution region attributes on page 723
Automatic overlay support

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.100 --override_visibility

Enables EXPORT and IMPORT directives in a steering file to override the visibility of a symbol.

Syntax

```
--override_visibility
```

Parameters

None.

Operation

By default:

- Only symbol definitions with STV_DEFAULT or STV_PROTECTED visibility can be exported.
- Only symbol references with STV_DEFAULT visibility can be imported.

When you specify `--override_visibility`, any global symbol definition can be exported and any global symbol reference can be imported.

Related information

--undefined_and_export=symbol on page 552
EXPORT steering file command on page 759
IMPORT steering file command on page 761

3.1.101 -Omax (armlink)

Enables maximum *Link-Time Optimization* (LTO).

Default

`-Omax` automatically enables the `--lto` and `--lto_level=Omax` options.

Syntax

```
-Omax
```

Parameters

None.

Operation

If you have object files that have been compiled with the `armclang` option `-Omax`, then you can link them using the `armlink` option `-Omax` to produce an image with maximum LTO.

Related information

--lto_level on page 501
--lto, --no_lto on page 498

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

-O (armclang) on page 203
Optimizing across modules with Link-Time Optimization

3.1.102 -Omin (armlink)

Enables the *Link-Time Optimization* (LTO) level `Omin` to produce an image with reduced code size.

Default

`-Omin` automatically enables the `--lto` and `--lto_level=Omin` options.

Syntax

```
-Omin
```

Parameters

None.

Operation

If you have object files that have been compiled with the `armclang` option `-Omin`, then you can link them using the `armlink` option `-Omin`.

Related information

--lto_level on page 501
--lto, --no_lto on page 498
-O (armclang) on page 203
Optimizing across modules with Link-Time Optimization

3.1.103 --pad=num

Enables you to set a value for padding bytes. The linker assigns this value to all padding bytes inserted in load or execution regions.

Syntax

```
--pad=<num>
```

Parameters

<num>

An integer, which can be given in hexadecimal format.

For example, setting `<num>` to `FF` might help to speed up ROM programming time. If `<num>` is greater than `FF`, then the padding byte is cast to a `char`, that is `(char)<num>`.

Operation

Padding is only inserted:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Within load regions. No padding is present between load regions.
- Between fixed execution regions (in addition to forcing alignment). Padding is not inserted up to the maximum length of a load region unless it has a fixed execution region at the top.
- Between sections to ensure that they conform to alignment constraints.

Related information

Input sections, output sections, regions, and program segments on page 572
Load view and execution view of an image on page 574

3.1.104 --paged

Enables Demand Paging mode to help produce ELF files that can be demand paged efficiently.

Syntax

```
--paged
```

Parameters

None.

Operation

A default page size of `0x8000` bytes is used. You can change this with the `--pagesize` command-line option.



Note

When linking with `--scatter`, all non-`ZI` execution regions in the scatter file must be root regions.

Related information

--pagesize=pagesize on page 514
--sysv on page 548
Linker support for creating demand-paged files on page 595
Alignment of regions to page boundaries on page 693

3.1.105 --pagesize=pagesize

Allows you to change the page size used when demand paging.

Default

The default value is `0x8000`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
--pagesize=<pagesize>
```

Parameters

<pagesize>

The page size in bytes.

Related information

--paged on page 514
Linker support for creating demand-paged files on page 595
Alignment of regions to page boundaries on page 693

3.1.106 --partial

Creates a partially-linked object that can be used in a subsequent link step.

Syntax

```
--partial
```

Parameters

None.

Restrictions

You cannot use `--partial` with `--scatter`.

Related information

Partial linking model overview on page 569

3.1.107 --pie

Specifies the *Position Independent Executable* (PIE) linking model.

Syntax

```
--pie
```

Parameters

None.

Restrictions

You must use this option with the `--fpic` and `--ref_pre_init` options.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--fpic on page 469
--bare_metal_pie on page 430
--ref_pre_init, --no_ref_pre_init on page 522

3.1.108 --piveneer, --no_piveneer

Enables or disables the generation of a veneer for a call from *Position Independent* (PI) code to absolute code.

Default

The default is `--piveneer`.

Syntax

```
--piveneer
--no_piveneer
```

Parameters

None.

Operation

When using `--no_piveneer`, an error message is produced if the linker detects a call from PI code to absolute code.

Related information

--inlineveneer, --no_inlineveneer on page 480
--veneershare, --no_veneershare on page 557
Generation of position independent to absolute veneers on page 600
Linker-generated veneers on page 597
Veneer sharing on page 598
Veneer types on page 599
Reuse of veneers when scatter-loading on page 600

3.1.109 --pixolib

Generates a *Position Independent eXecute Only* (PIXO) library.

Default

`--pixolib` is disabled by default.

Syntax

```
--pixolib
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Restrictions

Generation of PIXO libraries is only supported for Arm®v7-M targets.

When linking your application code with your PIXO library:

- The linker must not remove any unused sections from the PIXO library. You can ensure this with the armlink `--keep` command-line option.
- The `RW` sections with `SHT_NOBITS` and `SHT_PROGBITS` must be kept in the same order and relative offset for each PIXO library in the final image, as they were in the original PIXO libraries before linking the final image.

Operation

Use `--pixolib` to create a PIXO library, which is a relocatable library containing eXecutable Only code.

When creating the PIXO library, if you use `armclang` to invoke the linker, then `armclang` automatically passes the linker option `--pixolib` to `armlink`. If you invoke the linker separately, then you must use the `armlink` command-line option `--pixolib`. When creating a PIXO library, you must also provide a scatter file to the linker.

Each PIXO library must contain all the required standard library functions. Arm Compiler for Embedded 6 provides PIXO variants of the standard libraries based on Microlib. You must specify the required libraries on the command-line when creating your PIXO library. These libraries are location in the compiler installation directory under `/lib/pixolib/`.

The PIXO variants of the standard libraries have the naming format `<base>.<endian>`:

<base>

mc_wg
C library.

m_wgv
Math library for targets with hardware double precision floating-point support that is compatible with `vfpv5-d16`.

m_wgm
Math library for targets with hardware single precision floating-point support that is compatible with `fpv4-sp-d16`.

m_wgs
Math library for targets without hardware support for floating-point.

mf_wg
Software floating-point library. This library is required when:

- Using `printf()` to print floating-point values.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Using a math library that does not have all the required floating-point support in hardware. For example if your code has double precision floating-point operations but your target has `fpv4-sp-d16`, then the software floating-point library is used for the double-precision operations.

<endian>

l
Little endian

b
Big endian

Examples

This example shows the command-line invocations for compiling and linking in separate steps, to create a PIXO library from the source file `foo.c`.

```
armclang --target=arm-arm-none-eabi -march=armv7-m -mpixolib -c -o foo.o foo.c

armlink --pixolib --scatter=pixo.scf -o foo-pixo-library.o foo.o mc_wg.l
```

This example shows the command-line invocations for compiling and linking in a single step, to create a PIXO library from the source file `foo.c`.

```
armclang --target=arm-arm-none-eabi -march=armv7-m -mpixolib -Wl,--scatter=pixo.scf
 -o foo-pixo-library.o foo.c mc_wg.l
```

Related information

-mpixolib on page 189
--keep=section_id (armlink) on page 482
--startup=symbol, --no_startup on page 539
The Arm C Micro-Library

3.1.110 --predefine="string"

Enables commands to be passed to the preprocessor when preprocessing a scatter file.

You specify a preprocessor on the first line of the scatter file.

Syntax

```
--predefine="<string>"
```

Parameters

"<string>"
The command to pass to the preprocessor.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

You can use more than one `--predefine` option on the command-line.

You can also use the synonym `--pd="<string>"`.

Restrictions

Use this option with `--scatter`.

Example: Scatter file before preprocessing

The following example shows the scatter file contents before preprocessing.

```
#! armclang -E
lr1 BASE
{
    er1 BASE
    {
        *(+RO)
    }
    er2 BASE2
    {
        *(+RW+ZI)
    }
}
```

The following `armlink` command passes the command-line options `-DBASE=0x8000 -DBASE2=0x1000000` to the compiler to preprocess the scatter file:

```
armlink --predefine="-DBASE=0x8000" --predefine="-DBASE2=0x1000000" --
scatter=<filename>
```

Example: Scatter file after preprocessing

The following example shows how the scatter file looks after preprocessing:

```
lr1 0x8000
{
    er1 0x8000
    {
        *(+RO)
    }
    er2 0x1000000
    {
        *(+RW+ZI)
    }
}
```

Related information

Preprocessing a scatter file on page 695
--scatter=filename on page 529

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.111 --preinit, --no_preinit

Enables the linker to use a different image pre-initialization routine if required.

Default

The default for `<symbol>` is `__arm_preinit_`.

Syntax

```
--preinit=<symbol>
--no-preinit
```

Parameters

`<symbol>`

The name of the pre-initialization routine.

Operation

The linker adds a non-weak reference to `<symbol>` if a `.preinit_array` section is detected.

For `--preinit=__arm_preinit_` or `--cppinit=__cpp_initialize__aeabi_`, the linker processes R_ARM_TARGET1 relocations as R_ARM_REL32, because this is required by the `__arm_preinit` and `__cpp_initialize__aeabi_` functions. In all other cases R_ARM_TARGET1 relocations are processed as R_ARM_ABS32.

`--no_preinit` does not take a symbol argument.

Related information

--cppinit, --no_cppinit on page 441
--fpic on page 469
--ref_pre_init, --no_ref_pre_init on page 522
--bare_metal_pie on page 430

3.1.112 --privacy (armlink)

Modifies parts of an image to help protect your code.

Syntax

```
--privacy
```

Parameters

None.

Operation

The effect of this option is different for images and object files.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

When producing an executable image it removes local symbols from the output symbol table.

For object files built with the `--partial` option, this option:

- Changes section names to a default value, for example, changes code section names to `.text`.
- Keeps mapping symbols and build attributes in the symbol table.
- Removes those local symbols that can be removed without loss of functionality.

  Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.



Note

To help protect your code in images and objects that are delivered to third parties, use the `fromelf --privacy` command.

Related information

--locals, --no_locals on page 497
--partial on page 515
--privacy (fromelf) on page 816
--strip=option[,option,...] on page 823
Options to protect code in object files with fromelf

3.1.113 --ref_cpp_init, --no_ref_cpp_init

Enables or disables the adding of a reference to the C++ static object initialization routine in the Arm libraries.

Default

The default is `--ref_cpp_init`, which adds the default reference `__cpp_initialize__aeabi_`.

Syntax

```
--ref_cpp_init
--no-ref_cpp_init
```

Parameters

None.

Operation

To change the default reference from `__cpp_initialize__aeabi_`, use the `--cppinit` option.

Use `--no_ref_cpp_init` if you are not going to use the Arm libraries.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--cppinit, --no_cppinit on page 441
--preinit, --no_preinit on page 519

3.1.114 --ref_pre_init, --no_ref_pre_init

Allows the linker to add or not add references to the image pre-initialization routine in the Arm libraries. The default reference added is `__arm_preinit_`. To change this you can use `--preinit`.

Default

The default is `--no_ref_pre_init`.

Syntax

```
--ref_pre_init
--no_ref_pre_init
```

Parameters

None.

Related information

--fpic on page 469
--preinit, --no_preinit on page 519
--bare_metal_pie on page 430

3.1.115 --reloc

Creates a single relocatable load region with contiguous execution regions.



Note

This option is deprecated. The BPABI and the Base Platform models are no longer supported.

Syntax

```
--reloc
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

You cannot use `--reloc` with `--scatter`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

You cannot use this option with `--xo_base`.

Operation

Only use this option for legacy systems with the type of relocatable ELF images that conform to the *ELF for the Arm Architecture* specification. The generated image might not be compliant with the ELF for the Arm Architecture specification.

When relocated `MOVT` and `MOVW` instructions are encountered in an image being linked with `--reloc`, `armlink` produces the following additional dynamic tags:

`DT_RELA`
The address of a relocation table.

`DT_RELASZ`
The total size, in bytes, of the DT_RELA relocation table.

`DT_RELAENT`
The size, in bytes, of the DT_RELA relocation entry.

Related information

Type 1 image, one load region and contiguous execution regions on page 699
Type 3 image structure, multiple load regions and non-contiguous execution regions on page 587
ELF for the Arm Architecture

3.1.116 --remarks

Enables the display of remark messages, including any messages redesignated to remark severity using `--diag_remark`.


Note

The linker does not issue remarks by default.

Syntax

```
--remarks
```

Parameters

None.

Related information

--diag_remark=tag[,tag,...] (armlink) on page 451
--errors=filename on page 463

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.117 --remove, --no_remove

Enables or disables the removal of unused input sections from the image.

Default

The default is `--remove`.

If you specify `--shared` and `--sysv`, the default is `--no_remove`.

Syntax

```
--remove
--no_remove
```

Parameters

None.

Operation

An input section is considered used if it contains an entry point, or if it is referred to from a used section.

Use `--no_remove` when debugging to retain all input sections in the final image even if they are unused.

Use `--remove` with the `--keep` option to retain specific sections in a normal build.

Related information

--shared on page 532
--sysv on page 548
Elimination of unused sections on page 611
How the linker performs library searching, selection, and scanning on page 606
--keep=section_id (armlink) on page 482
Elimination of common section groups on page 610

3.1.118 --require-bti

Causes `armlink` to output an error when linking BTI with non-BTI user objects without the `--library_security=pacbti-m` option.

Syntax

```
--require-bti
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

When linking BTI with non-BTI user objects, `armlink` might or might not output a diagnostic message depending on what options are used:

- Without the `--require-bti` and `--library_security=pacbti-m` options, `armlink` outputs the warning:

  ```
  Warning: L6110W: Composition of BTI and non-BTI objects detected. The PACBTI-M
  library variant has been selected. Use --info=bti to print out the list of objects
  with their corresponding BTI mark.
  ```

- With the `--require-bti` option, but without the `--library_security=pacbti-m` option, `armlink` outputs warning `L6110W` as error `L6111E`:

  ```
  Error: L6111E: Composition of BTI and non-BTI objects detected. Use --info=bti to
  print out the list of objects with their corresponding BTI mark.
  ```

- With the `--library_security=pacbti-m` option, `armlink` does not output the warning or error message.

Use the `--info=bti` option to output a list of the BTI and non-BTI user objects in the link.

Related information

--info=topic[,topic,...] (armlink) on page 474
--check_pac_mismatch on page 440
--library_security=protection on page 492

3.1.119 --ro_base=address

Sets both the load and execution addresses of the region containing the RO output section at a specified address.

Default

If this option is not specified, and no scatter file is specified, the default is `--ro_base=0x8000`. If XO sections are present, then this is the default value used to place the ER_XO region.

When using `--shared`, the default is `--ro_base=0x0`.

Syntax

```
--ro_base=<address>
```

Parameters

`<address>`

Must be word-aligned.

Restrictions

You cannot use `--ro_base` with:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`--scatter`

Operation

If *eXecute-Only* (XO) sections are present, and you specify `--ro_base` without `--xo_base`, then an ER_XO execution region is created at the address specified by `--ro_base`. The ER_RO execution region immediately follows the ER_XO region.

Related information

--ropi on page 526
--rosplit on page 527
--rw_base=address on page 527
--xo_base=address on page 560
--zi_base=address on page 562

3.1.120 --ropi

Makes the load and execution region containing the RO output section position independent.

Syntax

```
--ropi
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

You cannot use `--ropi`:

- With `--fpic`, `--scatter`, or `--xo_base`.
- When an object file contains execute-only sections.

Operation

If this option is not used, the region is marked as absolute. Usually each read-only input section must be Read-Only Position Independent (ROPI). If this option is selected, the linker:

- Checks that relocations between sections are valid.
- Ensures that any code generated by the linker itself, such as interworking veneers, is ROPI.



Note

The linker gives a downgradable error if `--ropi` is used without `--rwpi` or `--rw_base`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--ro_base=address on page 525
--rosplit on page 527
--rw_base=address on page 527
--xo_base=address on page 560
--zi_base=address on page 562

3.1.121 --rosplit

Splits the default RO load region into two RO output sections.

Syntax

```
--rosplit
```

Parameters

None.

Restrictions

You cannot use `--rosplit` with `--scatter`.

Operation

The RO load region is split into the RO output sections:

- `RO-CODE`.
- `RO-DATA`.

Related information

--ro_base=address on page 525
--ropi on page 526
--rw_base=address on page 527
--xo_base=address on page 560
--zi_base=address on page 562

3.1.122 --rw_base=address

Sets the execution addresses of the region containing the RW output section at a specified address.

Syntax

```
--rw_base=<address>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<address>

Must be word-aligned.


Note

This option does not affect the placement of execute-only sections.

Restrictions

You cannot use `--rw_base` with `--scatter`.

Related information

--ro_base=address on page 525
--ropi on page 526
--rosplit on page 527
--xo_base=address on page 560
--zi_base=address on page 562

3.1.123 --rwpi

Makes the load and execution region containing the RW and ZI output section position independent.

Syntax

```
--rwpi
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

You cannot use `--rwpi`:

- With `--fpic --scatter`, or `--xo_base`.
- When an object file contains execute-only sections.

Operation

If this option is not used the region is marked as absolute. This option requires a value for `--rw_base`. If `--rw_base` is not specified, `--rw_base=0` is assumed. Usually each writable input section must be *Read/Write Position Independent* (RWPI).

If this option is selected, the linker:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Checks that the `PI` attribute is set on input sections to any read-write execution regions.
- Checks that relocations between sections are valid.
- Generates entries relative to the static base in the table `Region$$Table`. This table is used when regions are copied, decompressed, or initialized.

Related information

--shared on page 532
--sysv on page 548
--split on page 538
--scatter=filename on page 529
Region table format

3.1.124 --scanlib, --no_scanlib

Enables or disables scanning of the Arm libraries to resolve references.

Default

The default is `--scanlib`. However, if you specify `--shared`, then the default is `--no_scanlib`.

Syntax

```
--scanlib
--no_scanlib
```

Parameters

None.

Operation

Use `--no_scanlib` if you want to link your own libraries.

Related information

--stdlib on page 539

3.1.125 --scatter=filename

Creates an image memory map using the scatter-loading description that is contained in the specified file.

Syntax

```
--scatter=<filename>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<filename>

The name of a scatter file.

Operation

The scatter-loading description provides grouping and placement details of the various regions and sections in the image.

To modify the placement of any unassigned input sections when `.ANY` selectors are present, use the following command-line options with `--scatter`:

- `--any_contingency`.
- `--any_placement`.
- `--any_sort_order`.

You cannot use the `--scatter` option with:

- `--first`.
- `--last`.
- `--partial`.
- `--reloc`.
- `--ro_base`.
- `--ropi`.
- `--rosplit`.
- `--rw_base`.
- `--rwpi`.
- `--split`.
- `--xo_base`.
- `--zi_base`.

Related information

--any_contingency on page 425
--any_sort_order=order on page 427
Examples of using placement algorithms for .ANY sections on page 675
Preprocessing a scatter file on page 695
--first=section_id on page 467
--last=section_id on page 489
--ro_base=address on page 525
--ropi on page 526
--rosplit on page 527
--rw_base=address on page 527

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--rwpi on page 528
--split on page 538
--xo_base=address on page 560
--zi_base=address on page 562
--partial on page 515
--reloc on page 522
--shared on page 532
--sysv on page 548
Scatter-loading Features on page 643
Behavior when .ANY sections overflow because of linker-generated content on page 681

3.1.126 --scatterload-enabled, --no-scatterload-enabled

Enables or disables the generation of scatter-loaded images.

Syntax

Enable the generation of scatter-loaded images.

```
--scatterload-enabled
```

Disable the generation of scatter-loaded images.

```
--no-scatterload-enabled
```

Parameters

None.

Operation

Setting `--elf-output-format=gnu` implies `--no-scatterload-enabled`.

Related information

--elf-output-format on page 456
Building images that are compatible with third-party tools

3.1.127 --section_index_display=type

Changes the display of the index column when printing memory map output.

Default

The default is `--section_index_display=internal`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
--section_index_display=<type>
```

Parameters

<type>

One of the following:

cmdline

Alters the display of the map file to show the order that a section appears on the command-line. The command-line order is defined as `File.Object.Section` where:

- `Section` is the section index, `sh_idx`, of the `Section` in the `Object`.
- `Object` is the order that `Object` appears in the `File`.
- `File` is the order the `File` appears on the command line.

The order the `Object` appears in the `File` is only significant if the file is an `ar` archive.

internal

The index value represents the order in which the linker creates the section.

input

The index value represents the section index of the section in the original input file. This is useful when you want to find the exact section in an input object.

Operation

Use this option with `--map`.

Related information

--map, --no_map on page 504
--tiebreaker=option on page 550

3.1.128 --shared

Creates a *System V* (SysV) shared object.

Default

This option is disabled by default.

Syntax

```
--shared
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Restrictions

Not supported in the Keil® Microcontroller Development Kit (Keil® MDK).

Operation

You must use this option with `--fpic` and `--sysv`.

Note

By default, this option:

- Disables the scanning of the Arm C and C++ libraries to resolve references. Use the `--scanlib` option to enable the scanning of the Arm libraries.
- Disables unused section elimination. Use the `--remove` option to enable unused section elimination when building a shared object.
- Disables the adding of a reference to the C++ static object initialization routine in the Arm libraries. Use the `--ref_cpp_init` option to enable this feature.
- Changes the default value for `--ro_base` to `0x0000`.

Related information

--fpic on page 469
--import_unresolved, --no_import_unresolved on page 474
--ref_cpp_init, --no_ref_cpp_init on page 521
--remove, --no_remove on page 523
--soname=name on page 535
--sysv on page 548
SysV Shared Libraries and Executables on page 745

3.1.129 --show_cmdline (armlink)

Outputs the `armlink` command line.

Syntax

```
--show_cmdline
```

Parameters

None.

Operation

Shows the command line after processing by the tool, and can be useful to check:

- The command line a build system is using.
- How the tool 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.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The output is sent to the standard error stream (`stderr`).

Related information

--help (armlink) on page 472
--via=filename (armlink) on page 558

3.1.130 --show_full_path

Displays the full path name of an object in diagnostic messages and memory maps.

Syntax

```
--show_full_path
```

Parameters

None.

Operation

If the file representing object `obj` has the full path name `path/to/obj`, then the linker displays `path/to/obj` instead of `obj`.

Related information

--show_parent_lib on page 534
--show_sec_idx on page 535

3.1.131 --show_parent_lib

Displays the library name containing an object in diagnostic messages and memory maps.

Default

This option is on by default.

Syntax

```
--show_parent_lib
```

Parameters

None.

Operation

If an object `obj` comes from library `lib`, then this option displays `lib(obj)` instead of `obj`.

Related information

--show_full_path on page 534
--show_sec_idx on page 535

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.132 --show_sec_idx

Displays the section index, `sh_idx`, of section in the originating object.

Syntax

```
--show_sec_idx
```

Parameters

None.

Example

If section `sec` has section index 3, then it is displayed as `sec:3` in all diagnostic messages and memory maps.

Related information

--show_full_path on page 534
--show_parent_lib on page 534

3.1.133 --soname=name

Specifies the shared object runtime name that is used as the dependency name by any object that links against this shared object.

Syntax

```
--soname=<name>
```

Parameters

`<name>`

The runtime name of the shared object. The dependency name is stored in the resultant file.

Restrictions

This option is relevant only when used with `--shared`, and the default is the name of the shared object being generated.

Related information

SysV Shared Libraries and Executables on page 745

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.134 --sort=algorithm

Specifies the sorting algorithm used by the linker to determine the order of sections in an output image.


Note

Data sections are not included in the sort.

Default

The default algorithm is `--sort=Lexical`. In large region mode, the default algorithm is `--sort=AvgCallDepth`.

Syntax

```
--sort=<algorithm>
```

Parameters

`<algorithm>`

One of the following:

`Alignment`

Sorts input sections by ascending order of alignment value.

`AlignmentLexical`

Sorts input sections by ascending order of alignment value, then sorts lexically.

`AvgCallDepth`

Sorts all T32 code before A32 code and then sorts according to the approximated average call depth of each section in ascending order.

Use this algorithm to minimize the number of long branch veneers.


Note

The approximation of the average call depth depends on the order of input sections. Therefore, this sorting algorithm depends more on the order of input sections than using, for example, `RunningDepth`.

`BreadthFirstCallTree`

This is similar to the `CallTree` algorithm except that it uses a breadth-first traversal when flattening the Call Tree into a list.

`CallTree`

The linker flattens the call tree into a list containing the read-only code sections from all execution regions that have `CallTree` sorting enabled.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Sections in this list are copied back into their execution regions, followed by all the non read-only code sections, sorted lexically. Doing this ensures that sections calling each other are placed close together.


Note

This sorting algorithm depend less on the order of input sections than using either `RunningDepth` or `AvgCallDepth`.

`Lexical`

Sorts according to the name of the section and then by input order if the names are the same.

`LexicalAlignment`

Sorts input sections lexically, then by input order if the names are the same, and then by ascending order of alignment value.

`LexicalState`

Sorts T32 code before A32 code, then sorts lexically.

`List`

Provides a list of the available sorting algorithms. The linker terminates after displaying the list.

`ObjectCode`

Sorts code sections by tiebreaker. All other sections are sorted lexically. This is most useful when used with `--tiebreaker=cmdline` because it attempts to group all the sections from the same object together in the memory map.

`RunningDepth`

Sorts all T32 code before A32 code and then sorts according to the running depth of the section in ascending order. The running depth of a section s is the average call depth of all the sections that call s, weighted by the number of times that they call s.

Use this algorithm to minimize the number of long branch veneers.

Operation

The sorting algorithms conform to the standard rules, placing input sections in ascending order by attributes.

You can also specify sort algorithms in a scatter file for individual execution regions. Use the `SORTTYPE` keyword to do this.


Note

The `SORTTYPE` execution region attribute overrides any sorting algorithm that you specify with this option.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--tiebreaker=option on page 550
--largeregions, --no_largeregions on page 487
Execution region attributes on page 723
Section placement with the linker on page 590
Execution region descriptions on page 720

3.1.135 --split

Splits the default load region, that contains the RO and RW output sections, into separate load regions.

Syntax

```
--split
```

Parameters

None.

Restrictions

You cannot use `--split` with `--scatter`.

Operation

The default load region is split into the following load regions:

- One region containing the RO output section. The default load address is `0x8000`, but you can specify a different address with the `--ro_base` option.
- One region containing the RW and ZI output sections. The default load address is `0x0`, but you can specify a different address with the `--rw_base` option.

Both regions are root regions.

Considerations when execute-only sections are present

For images containing *eXecute-Only* (XO) sections, an XO execution region is placed at the address specified by `--ro_base`. The RO execution region is placed immediately after the XO region.

If you specify `--xo_base <address>`, then the XO execution region is placed at the specified address in a separate load region from the RO execution region.

Related information

--scatter=filename on page 529
--shared on page 532
--sysv on page 548
The structure of an Arm ELF image on page 570

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.136 --startup=symbol, --no_startup

Enables the linker to use alternative C libraries with a different startup symbol if required.

Default

The default is `--startup=__main`.

Syntax

```
--startup=<symbol>
--no_startup
```

Parameters

`<symbol>`

A symbol that identifies the beginning of the startup code.

Operation

The linker includes the C library startup code if there is a reference to a symbol that is defined by the C library startup code. This symbol reference is called the startup symbol. It is automatically created by the linker when it sees a definition of `main()`. The `--startup` option enables you to change this symbol reference.

- If the linker finds a definition of `main()` and does not find a definition of `<symbol>`, then it generates an error.
- If the linker finds a definition of `main()` and a definition of `<symbol>`, but no entry point is specified, then it generates a warning.

`--no_startup` does not add a reference.

Related information

--entry=location on page 460

3.1.137 --stdlib

Specifies the C++ library to use.


Note

This topic includes descriptions of [ALPHA] features. See Support level definitions.

Syntax

```
--stdlib=<library_option>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<library_option>

One of the following:

libc++

The standard C++ library.

threaded_libc++ [ALPHA]

The threaded standard C++ library.

Operation

C++ objects compiled with `armclang` and linked with `armlink` use libc++ by default.

Related information

C++ libraries and multithreading [ALPHA]

3.1.138 --strict

Instructs the linker to perform additional conformance checks, such as reporting conditions that might result in failures.

Syntax

```
--strict
```

Parameters

None.

Operation

`--strict` causes the linker to check for taking the address of:

- A non-interworking location from a non-interworking location in a different state.
- A RW location from a location that uses the static base register R9.
- A `STKCKD` function in an image that contains `USESV7` functions.
- A `~STKCKD` function in an image that contains `STKCKD` functions.



Note

`STKCKD` functions reserve register R10 for Stack Checking, `~STKCKD` functions use register R10 as variable register v7 and `USESV7` functions use register R10 as v7. See the *Procedure Call Standard for the Arm Architecture* (AAPCS) for more information about v7.

An example of a condition that might result in failure is taking the address of an interworking function from a non-interworking function.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--strict_flags, --no_strict_flags on page 541
--strict_ph, --no_strict_ph on page 541
--strict_relocations, --no_strict_relocations on page 543
--strict_symbols, --no_strict_symbols on page 543
--strict_visibility, --no_strict_visibility on page 544

3.1.139 --strict_flags, --no_strict_flags

Prevent or allow the generation of the EF_ARM_HASENTRY flag.

Default

The default is `--no_strict_flags`.

Syntax

```
--strict_flags
--no_strict_flags
```

Parameters

None.

Operation

The option `--strict_flags` prevents the EF_ARM_HASENTRY flag from being generated.

Related information

--strict on page 540
--strict_ph, --no_strict_ph on page 541
--strict_relocations, --no_strict_relocations on page 543
--strict_symbols, --no_strict_symbols on page 543
--strict_visibility, --no_strict_visibility on page 544

3.1.140 --strict_ph, --no_strict_ph

Enables or disables the sorting of the Program Header table entries.

Default

The default is `--strict_ph`.

Syntax

```
--strict_ph
--no_strict_ph
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

The linker writes the contents of load regions into the output ELF file in the order that load regions are written in the scatter file. Each load region is represented by one ELF program segment.

Program Header table entries are sorted in ascending virtual address order.

Use the `--no_strict_ph` command-line option to switch off the sorting of the Program Header table entries.

Related information

--strict on page 540
--strict_flags, --no_strict_flags on page 541
--strict_relocations, --no_strict_relocations on page 543
--strict_symbols, --no_strict_symbols on page 543
--strict_visibility, --no_strict_visibility on page 544

3.1.141 --strict_preserve8_require8

Enables the generation of the `armlink` diagnostic `L6238E` when a function that is not tagged as preserving eight-byte alignment of the stack calls a function that is tagged as requiring eight-byte alignment of the stack.

Syntax

```
--strict_preserve8_require8
```

Parameters

None.

Operation

This option controls only the instances of error `L6283E` that relate to the preserve eight-byte stack alignment and require eight-byte stack alignment relationship, not any other instances of that error.

When a function is known to preserve eight-byte alignment of the stack, `armclang` assigns the build attribute `Tag_ABI_align_preserved` to that function. However, the `armclang` integrated assembler does not automatically assign this attribute to assembly code.

By default, `armlink` does not check for the build attribute `Tag_ABI_align_preserved`. Therefore, when you specify `--strict_preserve8_require8`, and `armlink` generates error `L6238E`, then you must check that your assembly code preserves eight-byte stack alignment. If it does, then add the following directive to your assembly code:

```
    .eabi_attribute Tag_ABI_align_preserved, 1
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

L6238E

3.1.142 --strict_relocations, --no_strict_relocations

Enables you to ensure *Application Binary Interface* (ABI) compliance of legacy or third-party objects.

Default

The default is `--no_strict_relocations`.

Syntax

```
--strict_relocations
--no_strict_relocations
```

Parameters

None.

Operation

This option checks that branch relocation applies to a branch instruction bit-pattern. The linker generates an error if there is a mismatch.

Use `--strict_relocations` to instruct the linker to report instances of obsolete and deprecated relocations.

Relocation errors and warnings are most likely to occur if you are linking object files built with previous versions of the Arm tools.

Related information

--strict on page 540
--strict_flags, --no_strict_flags on page 541
--strict_ph, --no_strict_ph on page 541
--strict_symbols, --no_strict_symbols on page 543
--strict_visibility, --no_strict_visibility on page 544

3.1.143 --strict_symbols, --no_strict_symbols

Checks whether or not a mapping symbol type matches an ABI symbol type.

Default

The default is `--no_strict_symbols`.

Syntax

```
--strict_symbols
--no_strict_symbols
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

The option `--strict_symbols` checks that the mapping symbol type matches ABI symbol type. The linker displays a warning if the types do not match.

A mismatch can occur only if you have hand-coded your own assembler.

Example: Assembler code symbol and type in A32

In the following assembler code the symbol `sym` has type `STT_FUNC` and is A32:

```
    .section mycode,"x"
    .word sym + 4
    .code 32
    .type sym, "function"
sym:
    mov  r0, r0
    .code 16
    mov  r0, r0
    .end
```

The difference in behavior is the meaning of `.word sym + 4`. In pre-ABI linkers the state of the symbol is the state of the mapping symbol at that location. In this example, the state is T32. In ABI linkers the type of the symbol is the state of the location of symbol plus the offset.

Related information

--strict on page 540
--strict_flags, --no_strict_flags on page 541
--strict_ph, --no_strict_ph on page 541
--strict_relocations, --no_strict_relocations on page 543
--strict_visibility, --no_strict_visibility on page 544

3.1.144 --strict_visibility, --no_strict_visibility

Prevents or allows a hidden visibility reference to match against a shared object.

Default

The default is `--strict_visibility`.

Syntax

```
--strict_visibility
--no_strict_visibility
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

A linker is not permitted to match a symbol reference with `STT_HIDDEN` visibility to a dynamic shared object. Some older linkers might permit this.

Use `--no_strict_visibility` to permit a hidden visibility reference to match against a shared object.

Related information

--strict on page 540
--strict_flags, --no_strict_flags on page 541
--strict_ph, --no_strict_ph on page 541
--strict_relocations, --no_strict_relocations on page 543
--strict_symbols, --no_strict_symbols on page 543

3.1.145 --summary_stderr, --no_summary_stderr

Sends the output of `--info=summarysizes` to `stderr`.



Note

The output sent to `stderr` as a result of this option is not affected by use of `--errors=<filename>`.

Default

The default is `--no_summary_stderr`, unless you are using the Keil® Microcontroller Development Kit (Keil® MDK).

Keil MDK sets `--summary_stderr` as the default. If you are using Arm® Compiler for Embedded as part of Keil MDK, then you can set `--no_summary_stderr` to prevent the output being written to `stderr`.

Syntax

```
--summary_stderr
--no_summary_stderr
```

Parameters

None.

Operation

Use `--summary_stderr` to send the output of `--info=summarysizes` to `stderr`.

Use `--no_summary_stderr` to stop sending the output of `--info=summarysizes` to `stderr`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--errors=filename on page 463
--info=topic[,topic,...] (armlink) on page 474
Getting Image Details

3.1.146 --symbols, --no_symbols

Enables or disables the listing of each local and global symbol used in the link step, and its value.


Note

This does not include mapping symbols output to `stdout`. Use `--list_mapping_symbols` to include mapping symbols in the output.

Default

The default is `--no_symbols`.

Syntax

```
--symbols
--no_symbols
```

Parameters

None.

Related information

--list_mapping_symbols, --no_list_mapping_symbols on page 496

3.1.147 --symdefs=filename

Creates a file containing the global symbol definitions from the output image.

Default

By default, all global symbols are written to the symdefs file. If a symdefs file called `<filename>` already exists, the linker restricts its output to the symbols already listed in this file.


Note

If you do not want this behavior, be sure to delete any existing symdefs file before the link step.

Syntax

```
--symdefs=<filename>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<filename>

The name of the text file to contain the global symbol definitions.

Operation

If `<filename>` is specified without path information, the linker searches for it in the directory where the output image is being written. If it is not found, it is created in that directory.

You can use the symbol definitions file as input when linking another image.

Related information

Access symbols in another image on page 635

3.1.148 --symver_script=filename

Enables implicit symbol versioning.

Syntax

```
--symver_script=<filename>
```

Parameters

<filename>

A symbol version script.

Related information

Symbol versioning on page 756

3.1.149 --symver_soname

Enables implicit symbol versioning to force static binding.

Default

This is the default if you are generating a *Base Platform Application Binary Interface* (BPABI) compatible executable file but where you do not specify a version script with the option `--symver_script`.

Syntax

```
--symver_soname
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Restrictions

Not supported for AArch64 state.

Operation

Where a symbol has no defined version, the linker uses the shared object name (`SONAME`) contained in the file being linked.

Related information

Symbol versioning on page 756

3.1.150 --sysv

Creates a *System V* (SysV) formatted ELF executable file.

Default

This option is disabled by default.

Syntax

```
--sysv
```

Parameters

None.

Restrictions

The following restrictions apply:

- Arm® Compiler for Embedded does not support dynamic linking when targeting Cortex®-M processors.
- You cannot use this option if an object file contains execute-only sections.
- The following regions are not allowed in a scatter file:
  - `ARM_LIB_STACK`
  - `ARM_LIB_HEAP`
  - `ARM_LIB_STACKHEAP`

  Use the `--bare_metal_sysv` option to allow these regions to be used.

Operation

ELF files produced with the `--sysv` option are demand-paged compliant.

When linking with `--scatter`, all non-`ZI` execution regions in the scatter file must be root regions.

Related information

--bare_metal_sysv on page 430

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--remove, --no_remove on page 523
--fpic on page 469
--import_unresolved, --no_import_unresolved on page 474
SysV linking model overview on page 569
Linker support for creating demand-paged files on page 595
IMPORT steering file command on page 761
SysV Shared Libraries and Executables on page 745
SysV Dynamic Linking
Bare-metal Position Independent Executables

3.1.151 --tailreorder, --no_tailreorder

Moves tail calling sections immediately before their target, if possible, to optimize the branch instruction at the end of a section.

Default

The default is `--no_tailreorder`.

Syntax

```
--tailreorder
--no_tailreorder
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

The linker:

- Can only move one tail calling section for each tail call target. If there are multiple tail calls to a single section, the tail calling section with an identical section name is moved before the target. If no section name is found in the tail calling section that has a matching name, then the linker moves the first section it encounters.
- Cannot move a tail calling section out of its execution region.
- Does not move tail calling sections before inline veneers.

Operation

A tail calling section is a section that contains a branch instruction at the end of the section. The branch must have a relocation that targets a function at the start of a section.

Related information

Linker reordering of tail calling sections on page 618
--branchnop, --no_branchnop on page 434

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

About branches that optimize to a NOP on page 618

3.1.152 --tiebreaker=option

A tiebreaker is used when a sorting algorithm requires a total ordering of sections. It is used by the linker to resolve the order when the sorting criteria results in more than one input section with equal properties.

Default

The default option is `--tiebreaker=creation`.

Syntax

```
--tiebreaker=<option>
```

Parameters

`<option>`

One of:

`creation`

The order that the linker creates sections in its internal section data structure.

When the linker creates an input section for each ELF section in the input objects, it increments a global counter. The value of this counter is stored in the section as the creation index.

The creation index of a section is unique apart from the special case of inline veneers.

`cmdline`

The order that the section appears on the linker command-line. The command-line order is defined as `File.Object.Section` where:

- `Section` is the section index, `sh_idx`, of the `Section` in the `Object`.
- `Object` is the order that `Object` appears in the `File`.
- `File` is the order the `File` appears on the command line.

The order the `Object` appears in the `File` is only significant if the file is an `ar` archive.

This option is useful if you are doing a binary difference between the results of different links, link1 and link2. If link2 has only small changes from link1, then you might want the differences in one source file to be localized. In general, creation index works well for objects, but because of the multiple pass selection of members from libraries, a small difference such as calling a new function can result in a different order of objects and therefore a different tiebreaker. The command-line index is more stable across builds.

Use this option with the `--scatter` option.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--sort=algorithm on page 535
--map, --no_map on page 504
--any_sort_order=order on page 427

3.1.153 --unaligned_access, --no_unaligned_access (armlink)

Enable or disable unaligned accesses to data on Arm architecture-based processors.

Default

The default is `--unaligned_access`.

Syntax

```
--unaligned_access
--no_unaligned_access
```

Parameters

None.

Operation

When using `--no_unaligned_access`, the linker:

- Does not select objects from the Arm C library that allow unaligned accesses.
- Gives an error message if any input object allows unaligned accesses.


Note

You can downgrade this error message.

Related information

-munaligned-access, -mno-unaligned-access on page 197
Alignment support in Arm Compiler for Embedded 6

3.1.154 --undefined=symbol

Prevents the removal of a specified symbol if it is undefined.

Syntax

```
--undefined=<symbol>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<symbol>

The symbol to not remove.

Operation

Causes the linker to:

1. Create a symbol reference to the specified symbol name.
2. Issue an implicit `--keep=<symbol>` to prevent any sections brought in to define that symbol from being removed.

Related information

--undefined_and_export=symbol on page 552
--keep=section_id (armlink) on page 482

3.1.155 --undefined_and_export=symbol

Prevents the removal of a specified symbol if it is undefined, and pushes the symbol into the dynamic symbol table.

Syntax

```
--undefined_and_export=<symbol>
```

Parameters

<symbol>

The symbol to not remove.

Operation

Causes the linker to:

1. Create a symbol reference to the specified symbol name.
2. Issue an implicit `--keep=<symbol>` to prevent any sections brought in to define that symbol from being removed.
3. Add an implicit `EXPORT <symbol>` to push the specified symbol into the dynamic symbol table.

Considerations

Be aware of the following when using this option:

- It does not change the visibility of a symbol unless you specify the `--override_visibility` option.
- A warning is issued if the visibility of the specified symbol is not high enough.
- A warning is issued if the visibility of the specified symbol is overridden because you also specified the `--override_visibility` option.
- Hidden symbols are not exported unless you specify the `--override_visibility` option.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--override_visibility on page 511
--undefined=symbol on page 551
--keep=section_id (armlink) on page 482
EXPORT steering file command on page 759

3.1.156 --unresolved=symbol

Takes each reference to an undefined symbol and matches it to the global definition of the specified symbol.

Syntax

```
--unresolved=<symbol>
```

Parameters

<symbol>

Must be both defined and global, otherwise it appears in the list of undefined symbols and the link step fails.

Operation

This option is particularly useful during top-down development, because it enables you to test a partially-implemented system by matching each reference to a missing function to a dummy function.

Related information

--undefined=symbol on page 551
--undefined_and_export=symbol on page 552

3.1.157 --use_definition_visibility

Enables the linker to use the visibility of the definition in preference to the visibility of a reference when combining symbols.

Syntax

```
--use_definition_visibility
```

Parameters

None.

Operation

When the linker combines global symbols the visibility of the symbol is set with the strictest visibility of the symbols being combined. Therefore, a symbol reference with STV_HIDDEN visibility

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

combined with a definition with STV_DEFAULT visibility results in a definition with STV_HIDDEN visibility.

For example, a symbol reference with STV_HIDDEN visibility combined with a definition with STV_DEFAULT visibility results in a definition with STV_DEFAULT visibility.

This can be useful when you want a reference to not match a Shared Library, but you want to export the definition.


Note

This option is not ELF-compliant and is disabled by default. To create ELF-compliant images, you must use symbol references with the appropriate visibility.

3.1.158 --userlibpath=pathlist

Specifies a list of paths that the linker is to use to search for user libraries.

Syntax

```
--userlibpath=<pathlist>
```

Parameters

<pathlist>

A comma-separated list of paths that the linker is to use to search for the required libraries. Do not include spaces between the comma and the path name when specifying multiple path names, for example, <path1>,<path2>,<path3>,...,<pathn>.

Related information

--libpath=pathlist on page 491

3.1.159 --veneerinject, --no_veneerinject

Enables or disables the placement of veneers outside of the sorting order for the Execution Region.

Default

The default is --no_veneerinject. The linker automatically switches to large region mode if it is required to successfully link the image. If large region mode is turned off with --no_largeregions then only --veneerinject is turned on if it is required to successfully link the image.


Note

--veneerinject is the default for large region mode.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
--veneerinject
--no_veneerinject
```

Parameters

None.

Operation

Use `--veneerinject` to allow the linker to place veneers outside of the sorting order for the Execution Region. This option is a subset of the `--largeregions` command. Use `--veneerinject` if you want to allow the veneer placement behavior described, but do not want to implicitly set the `--api` and `--sort=AvgCallDepth`.

Use `--no_veneerinject` to allow the linker use the sorting order for the Execution Region.

Use `--veneer_inject_type` to control the strategy the linker uses to place injected veneers.

The following command-line options allow stable veneer placement with large Execution Regions:

```
--veneerinject --veneer_inject_type=pool --sort=lexical
```

Related information

--largeregions, --no_largeregions on page 487
--veneer_inject_type=type on page 555
--api, --no_api on page 428
--sort=algorithm on page 535

3.1.160 --veneer_inject_type=type

Controls the veneer layout when `--largeregions` mode is on.

Syntax

```
--veneer_inject_type=<type>
```

Parameters

`<type>`

One of:

`individual`

The linker places veneers to ensure they can be reached by the largest amount of sections that use the veneer. Veneer reuse between execution regions is permitted. This type minimizes the number of veneers that are required but disrupts the structure of the image the most.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

pool

The linker:

1. Collects veneers from a contiguous range of the execution region.
2. Places all the veneers generated from that range into a pool.
3. Places that pool at the end of the range.

A large execution region might have more than one range and therefore more than one pool. Although this type has much less impact on the structure of image, it has fewer opportunities for reuse. This is because a range of code cannot reuse a veneer in another pool. The linker calculates the range based on the presence of branch instructions that the linker predicts might require veneers. A branch is predicted to require a veneer when either:

- A state change is required.
- The distance from source to target plus a contingency greater than the branch range.

You can set the size of the contingency with the `--veneer_pool_size=<size>` option. By default the contingency size is set to 102400 bytes. The `--info=veneerpools` option provides information on how the linker has placed veneer pools.

Restrictions

You must use `--largeregions` with this option.

Related information

--info=topic[,topic,...] (armlink) on page 474
--veneerinject, --no_veneerinject on page 554
--veneer_pool_size=size on page 556
--largeregions, --no_largeregions on page 487

3.1.161 --veneer_pool_size=size

Sets the contingency size for the veneer pool in an execution region.

Default

The default size is 102400 bytes.

Syntax

```
--veneer_pool_size=<size>
```

Parameters

<size>

The size in bytes.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--veneer_inject_type=type on page 555

3.1.162 --veneershare, --no_veneershare

Enables or disables veneer sharing. Veneer sharing can cause a significant decrease in image size.

Default

The default is `--veneershare`.

Syntax

```
--veneershare
--no_veneershare
```

Parameters

None.

Related information

--inlineveneer, --no_inlineveneer on page 480
--piveneer, --no_piveneer on page 516
Veneer sharing on page 598
Linker-generated veneers on page 597
Veneer types on page 599
Generation of position independent to absolute veneers on page 600
--crosser_veneershare, --no_crosser_veneershare on page 446

3.1.163 --verbose

Prints detailed information about the link operation, including the objects that are included and the libraries from which they are taken.

Syntax

```
--verbose
```

Parameters

None.

Operation

This output is particular useful for tracing undefined symbols reference or multiply defined symbols. Because this output is typically quite long, you might want to use this command with the `--list=<filename>` command to redirect the information to `<filename>`.

Use `--verbose` to output diagnostics to `stdout`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--list=filename on page 495
--muldefweak, --no_muldefweak on page 508
--unresolved=symbol on page 553

3.1.164 --version_number (armlink)

Displays the version of `armlink` that you are using.

Syntax

```
--version_number
```

Parameters

None.

Operation

`armlink` displays the version number in the format `<Mmmuuxx>`, where:

- `<M>` is the major version number, `6`.
- `<mm>` is the minor version number.
- `<uu>` is the update number.
- `<xx>` is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions.

Related information

--help (armlink) on page 472
--vsn (armlink) on page 559

3.1.165 --via=filename (armlink)

Reads an additional list of input filenames and tool options from `<filename>`.

Syntax

```
--via=<filename>
```

Parameters

`<filename>`

The name of a via file containing options to be included on the command line.

Operation

You can enter multiple `--via` options on the `armasm`, `armlink`, `fromelf`, and `armar` command lines. You can also include the `--via` options within a via file.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Via File Syntax on page 1082
Via file syntax rules on page 1083

3.1.166 --vsn (armlink)

Displays the version information and the license details.

Syntax

```
--vsn
```

Parameters

None.

Operation

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm® Compiler for Embedded tool you are using. If you need to access the version in other tools or scripts, for example in build scripts, use the output from `--version_number`.

Table 3-5: Arm Compiler for Embedded Version Information

Field	Description
`Product: <Product Name>`	The product name, which depends on the currently activated license or suite that contains Arm Compiler for Embedded as a component, such as: • Arm *Software Success Kit* (SSK) with an SSK license entitlement. • Arm Development Studio. • Arm® Keil® *Microprocessor Development Kit* (MDK). For example, in Arm Development Studio, `--vsn` lists the product as: `Arm Development Studio <edition> Edition <version>`
`Component: <component>`	The version of Arm Compiler for Embedded. The version is made up of the major, minor, and update number: `Arm Compiler for Embedded <Major Version>.<Minor Version>.<Update Version>`
`Tool: <tool name> [<tool identifier>]`	The tool name and tool identifier. The tool identifier must not be used to identify versions of Arm Compiler for Embedded. The tool identifier is different for each operating system.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: armlink --vsn output for Arm Development Studio

The `armlink` output for Arm Compiler for Embedded 6.23 on Linux x86 with Arm Development Studio Gold Edition 2024.0:

```
Product: Arm Development Studio Gold Edition 2024.0
Component: Arm Compiler for Embedded 6.23
Tool: armlink [<toolid>]
```

Related information

--help (armlink) on page 472
--version_number (armlink) on page 558

3.1.167 --xo_base=address

Specifies the base address of an *eXecute-Only* (XO) execution region.

Syntax

```
--xo_base=<address>
```

Parameters

<address>

Must be word-aligned.

Restrictions

You can use `--xo_base` only with the bare-metal linking model.


Note

XO memory is supported only for Arm®v6-M, Armv7-M, and Armv8-M architectures. However, XO is not supported on Armv6-M for any form of position independent code.

You cannot use `--xo_base` with:

- `--reloc`.
- `--ropi`.
- `--rwpi`.
- `--scatter`.
- `--shared`.
- `--sysv`.

Operation

When you specify `--xo_base`:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- XO sections are placed in a separate load and execution region, at the address specified.
- No ER_XO region is created when no XO sections are present.

Related information

--ro_base=address on page 525
--ropi on page 526
--rosplit on page 527
--rw_base=address on page 527
--zi_base=address on page 562

3.1.168 --xref, --no_xref

Lists to stdout all cross-references between input sections.

Default

The default is `--no_xref`.

Syntax

```
--xref
--no_xref
```

Parameters

None.

Related information

--xrefdbg, --no_xrefdbg on page 561
--xreffrom, --xrefto on page 562

3.1.169 --xrefdbg, --no_xrefdbg

Lists to stdout all cross-references between input debug sections.

Default

The default is `--no_xrefdbg`.

Syntax

```
--xrefdbg
--no_xrefdbg
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--xref, --no_xref on page 561
--xreffrom, --xrefto on page 562

3.1.170 --xreffrom, --xrefto

Lists to stdout cross-references from and to input sections.

Syntax

```
--xreffrom=<object>(<section>)
--xrefto=<object>(<section>)
```

Parameters

<object>

The object file containing the input section <section>

<section>

The input section <section>.

Operation

This option lists to stdout cross-references:

- From input <section> in <object> to other input sections.
- To input <section> in <object> from other input sections.

This is a useful subset of the listing produced by the --xref linker option if you are interested in references from or to a specific input section. You can have multiple occurrences of this option to list references from or to more than one input section.


Note

On Unix systems your shell typically requires the parentheses to be escaped with backslashes. Alternatively, enclose the complete section specifier in double quotes, for example:

```
--xreffrom="init.o(init)"
```

Related information

--xref, --no_xref on page 561
--xrefdbg, --no_xrefdbg on page 561

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.1.171 --zi_base=address

Specifies the base address of an ER_ZI execution region.

Syntax

```
--zi_base=<address>
```

Parameters

<address>

Must be word-aligned.


Note

This option does not affect the placement of execute-only sections.

Restrictions

The linker ignores `--zi_base` if one of the following options is also specified:

- `--reloc`.
- `--rwpi`.
- `--split`.
- `--sysv`.

You cannot use `--zi_base` with `--scatter`.

Related information

--ro_base=address on page 525
--ropi on page 526
--rosplit on page 527
--rw_base=address on page 527
--xo_base=address on page 560

3.1.172 Examples for the armlink --elf-output-format=gnu command-line option

These examples show scatter files that are incompatible with `--elf-output-format=gnu` and a typical use case.

For more information about the `--elf-output-format` command-line option, see --elf-output-format.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Scatter files that contain a load region with non-contiguous execution regions

Create the file `test.c` containing the following C code:

```
volatile unsigned int a, b;

int main(void)
{
          return a + b;
}
```

The following scatter file `test.scat` is incompatible with the `--elf-output-format=gnu` option, because it has only one load region:

```
LOAD_RO 0x0 {
    EXEC_RO +0 {
        *(+RO)
    }
    EXEC_RW 0x20000000 {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }

    ARM_LIB_STACK 0x20001000 EMPTY -0x400  ; Stack region growing down
    {
    }
}
```

To compile and link `test.c`, enter:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m33 -c test.c -o test.o
armlink --keep=* --scatter=test.scat --elf-output-format=gnu test.o -o test.axf
```

The following warning is output:

```
Warning: L6873W: LOAD_RO has multiple non-contiguous child regions {EXEC_RW,
 ARM_LIB_STACK}
Finished: 0 information, 1 warning and 0 error messages.
```

Example: Scatter files that contain a load region with an execution region after RW or ZI

```
LR1 0x0
{
    * (vectors.o)
}

LR2 0x80000000
{
    ER_RO +0
    {
        * (+RO)
    }
    ER_RW +0
    {
        * (+RW)
    }
    ER_ZI +0
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    {
        * (+ZI)
    }
    ER_SPECIFIC +0 FIXED
    {
        *.o (mySpecificSection)
    }
    ARM_LIB_STACKHEAP +0 EMPTY 0x8000
    {
    }
}
```

To compile and link `test.c`, enter:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m33 -c test.c -o test.o
armlink --keep=* --scatter=test.scat --elf-output-format=gnu test.o -o test.axf
```

For this example, the linker is unable to report an appropriate error or warning. However, you can observe the error in the program header for LR2:

```
** Program header #<num> (PT_LOAD) [<attributes>]
    Size : <...> bytes (<Y> bytes in file)
    Virtual address: 0x80000000 (Alignment 4)

...

** Section #4 'ER_ZI' (SHT_NOBITS) [<attributes>]
    Size   : <...> bytes (alignment 4)
    Address: <0x80000000 + Y>

** Section #5 'ER_SPECIFIC' (SHT_PROGBITS) [<attributes>]
    Size   : <...> bytes (alignment 4)
    Address: <0x80000000 + Z>
```

The program header for LR2 reports a size of Y PT_LOAD bytes in file. When compared with the ELF sections and we add size Y to the program header base address we can see that the PT_LOAD of LR2 ends at the start of ER_ZI section. As such, the program header does not contain the ER_SPECIFIC execution region. When the output ELF image from linking with this scatter file is used with an ELF loader, the ELF loader does not initialize the ER_SPECIFIC execution region. Failure to initialize the ER_SPECIFIC execution region can result in unexpected run-time behavior.

To avoid this issue, modify the scatter file such that for load region LR2 that contains RW or ZI data, one of the the following is true:

- Move all execution regions within LR2 that contain RW or ZI data to a separate load region.
- All execution regions within LR2 that contain RW or ZI data are placed at a higher address than any other execution region within LR2.
- All execution regions containing RW or ZI data within LR2 that are placed at a lower address than any other execution region within LR2 have the NOCOMPRESS and ZEROPAD attributes. However, this modification might result in a larger loadable binary file.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Typical use case for --elf-output-format=gnu

Modify the scatter file `test.scat` from the incompatible example to provide a second load region for the RW data, `LOAD_RW`:

```
LOAD_RO 0x0 {
    EXEC_RO +0 {
        *(+RO)
    }
}
LOAD_RW 0x20000000 0x2000 {
    EXEC_RW 0x20000000 {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
    ARM_LIB_STACK 0x20001000 EMPTY -0x400  ; Stack region growing down
    {
    }
}
```

Compile and link `test.c` from the previous example:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m33 -c test.c -o test.o
armlink --keep=* --scatter=test.scat test.o -o test.axf
```

To view the sections in the `test.axf` image, enter:

```
fromelf -v test.axf
```

Pipe the output to `test.txt` using the format for your OS.

Link again with the command:

```
armlink --keep=* --scatter=test.scat --elf-output-format=gnu test.o -o testgnu.axf
```


Note

With the `--elf-output-format=gnu` option, `armlink` currently generates the following warning, which you can ignore:

```
Warning: L6874W: Load Region LOAD_RW requires that the first exec
 region to be root 1, contains Read-Only Data 0, and not contain
 overlay regions 0
Finished: 0 information, 1 warning and 0 error messages.
```

You can use `--diag_suppress=L6874` to suppress this warning.

View the sections in the `testgnu.axf` image:

```
fromelf -v testgnu.axf
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

In this example, `Program header #1` shows that the load address for the data is `0x20000000`.

3.2 Linking Models Supported by armlink

Describes the linking models supported by the Arm linker, `armlink`.

3.2.1 Overview of linking models

A linking model is a group of command-line options and memory maps that control the behavior of the linker.

The linking models supported by `armlink` are:

Bare-metal

This model does not target any specific platform. It enables you to create an image with your own custom operating system, memory map, and, application code if required. Some limited dynamic linking support is available. You can specify additional options depending on whether or not a scatter file is in use.

Bare-metal Position Independent Executable

This model produces a bare-metal *Position Independent Executable* (PIE). This is an executable that does not need to be executed at a specific address but can be executed at any suitably aligned address. All objects and libraries linked into the image must be compiled to be position independent.

Partial linking

This model produces a relocatable ELF object suitable for input to the linker in a subsequent link step. The partial object can be used as input to another link step. The linker performs limited processing of input objects to produce a single output object.

You can combine related options in each model to tighten control over the output.

Related information

Bare-metal linking model overview on page 567
Partial linking model overview on page 569
SysV Shared Libraries and Executables on page 745

3.2.2 Bare-metal linking model overview

The bare-metal linking model focuses on the conventional embedded market where the whole program, possibly including a *Real-Time Operating System* (RTOS), is linked in one pass.

The linker can make very few assumptions about the memory map of a bare-metal system. Therefore, you must use the scatter-loading mechanism if you want more precise control. Scatter-loading allows different regions in an image memory map to be placed at addresses other than

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

at their natural address. Such an image is a relocatable image, and the linker must adjust program addresses and resolve references to external symbols.

By default, the linker attempts to resolve all the relocations statically. However, it is also possible to create a position independent or relocatable image. Such an image can be executed from different addresses and have its relocations resolved at load or run-time. You can use a dynamic model to create relocatable images. A position independent image does not require a dynamic model.

With the bare-metal model, you can:

- Identify the regions to be relocated or are position independent using a scatter file or command-line options.
- Identify the symbols to be imported and exported using a steering file.

You can use `--scatter=<file>` with this model.

You can use the following options when scatter-loading is not used:

- `--reloc` (not supported for AArch64 state).
- `--ro_base=<address>`.
- `--ropi`.
- `--rosplit`.
- `--rw_base=<address>`.
- `--rwpi`.
- `--split`.
- `--xo_base=<address>`.
- `--zi_base`.


Note

`--xo_base` cannot be used with `--ropi` or `--rwpi`.

Related information

--xo_base=address on page 560
Methods of specifying an image memory map with the linker on page 576
--edit=file_list on page 455
--reloc on page 522
--ro_base=address on page 525
--ropi on page 526
--rosplit on page 527
--rw_base=address on page 527
--rwpi on page 528
--scatter=filename on page 529

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--split on page 538
--zi_base=address on page 562
Linker Steering File Command Reference on page 759

3.2.3 Partial linking model overview

The partial linking model produces a single output file that can be used as input to a subsequent link step.

Partial linking:

- Eliminates duplicate copies of debug sections.
- Merges the symbol tables into one.
- Leaves unresolved references unresolved.
- Merges common data (COMDAT) groups.
- Generates a single object file that can be used as input to a subsequent link step.

If the linker finds multiple entry points in the input files it generates an error because the single output file can have only one entry point.

To link with this model, use the `--partial` command-line option.


Note

If you use partial linking, you cannot refer to the original objects by name in a scatter file. Therefore, you might have to update your scatter file.

Related information

Edit the symbol tables with a steering file on page 639
Steering file format on page 640
Linker Steering File Command Reference on page 759
--edit=file_list on page 455
--partial on page 515

3.2.4 SysV linking model overview

The *System V* (SysV) model produces SysV shared objects and executables.

To link with this model and build a SysV executable, use the `--sysv` command-line option.

To build a SysV shared object, use `--sysv`, `--shared`, and `--fpic` options.

Be aware of the following:

- By default, the model assumes that shared objects can throw an exception.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- When building a SysV shared object, scanning of the Arm C and C++ libraries to resolve references is disabled by default. Use the `--scanlib` option to re-enable scanning of the Arm libraries.

Related information

SysV linking model on page 746

3.3 Image Structure and Generation

Describes the image structure and the functionality available in the Arm linker, `armlink`, to generate images.

3.3.1 The structure of an Arm ELF image

An Arm ELF image contains sections, regions, and segments, and each link stage has a different view of the image.

The structure of an image is defined by the:

- Number of its constituent regions and output sections.
- Positions in memory of these regions and sections when the image is loaded.
- Positions in memory of these regions and sections when the image executes.

3.3.1.1 Views of the image at each link stage

Each link stage has a different view of the image.

The image views are:

ELF object file view (linker input)

The ELF object file view comprises input sections. The ELF object file can be:

- A relocatable file that holds code and data suitable for linking with other object files to create an executable or a shared object file.
- A shared object file that holds code and data.

Linker view

The linker has two views for the address space of a program that become distinct in the presence of overlaid, position independent, and relocatable program fragments (code or data):

- The load address of a program fragment is the target address that the linker expects an external agent such as a program loader, dynamic linker, or debugger to copy the fragment from the ELF file. This might not be the address at which the fragment executes.
- The execution address of a program fragment is the target address where the linker expects the fragment to reside whenever it participates in the execution of the program.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If a fragment is position independent or relocatable, its execution address can vary during execution.

ELF image file view (linker output)

The ELF image file view comprises program segments and output sections:

- A load region corresponds to a program segment.
- An execution region contains one or more of the following output sections:
  - RO section.
  - RW section.
  - XO section.
  - ZI section.

One or more execution regions make up a load region.



Note

With `armlink`, the maximum size of a program segment is 2GB.

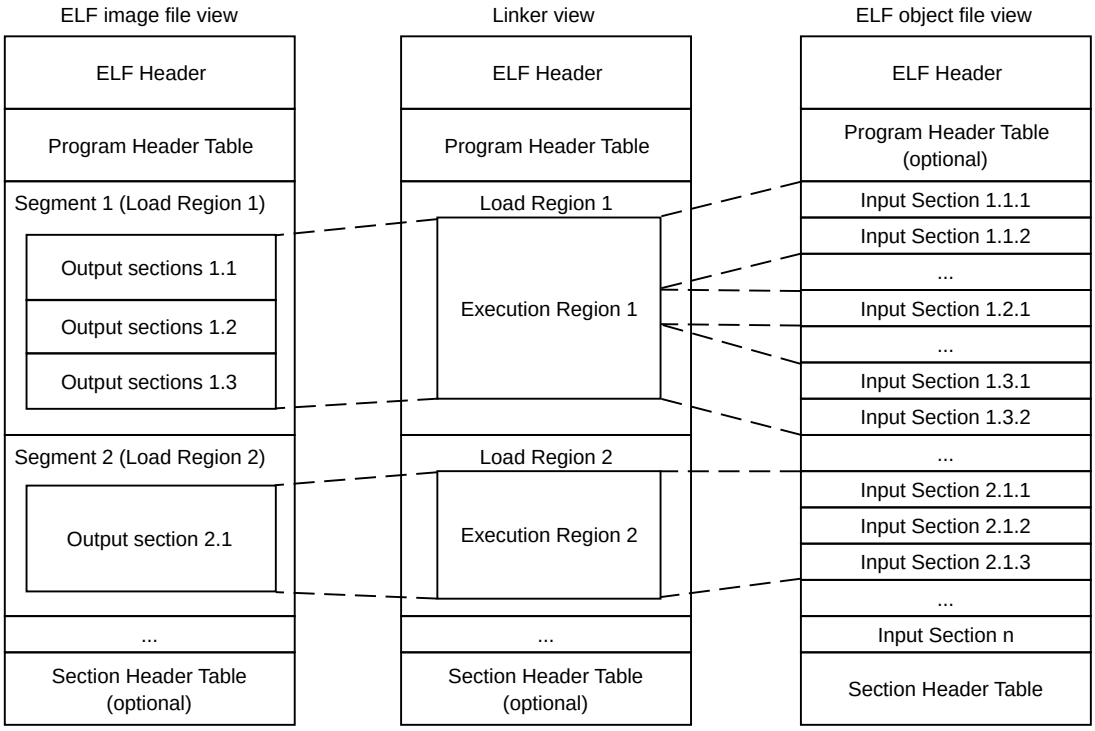
When describing a memory view:

- The term root region means a region that has the same load and execution addresses.
- Load regions are equivalent to ELF segments.

The following figure shows the relationship between the views at each link stage:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 3-1: Relationship between sections, regions, and segments



3.3.1.2 Input sections, output sections, regions, and program segments

An object or image file is constructed from a hierarchy of input sections, output sections, regions, and program segments.

Input section

An input section is an individual section from an input object file. It contains code, initialized data, or describes a fragment of memory that is not initialized or that must be set to zero before the image can execute. These properties are represented by attributes such as RO, RW, XO, and ZI. These attributes are used by `armlink` to group input sections into bigger building blocks called output sections and regions.

Output section

An output section is a group of input sections that have the same RO, RW, XO, or ZI attribute, and that are placed contiguously in memory by the linker. An output section has the same attributes as its constituent input sections. Within an output section, the input sections are sorted according to the section placement rules.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Region

A region contains up to three output sections depending on the contents and the number of sections with different attributes. By default, the output sections in a region are sorted according to their attributes:

- If no XO output sections are present, then the RO output section is placed first, followed by the RW output section, and finally the ZI output section.
- If all code in the execution region is execute-only, then an XO output section is placed first, followed by the RW output section, and finally the ZI output section.

A region typically maps onto a physical memory device, such as ROM, RAM, or peripheral. You can change the order of output sections using scatter-loading.

Program segment

A program segment corresponds to a load region and contains execution regions. Program segments hold information such as text and data.


Note

With `armlink`, the maximum size of a program segment is 2GB.


Note

XO memory is supported only for Arm®v6-M, Armv7-M, and Armv8-M architectures. However, XO is not supported on Armv6-M for any form of position independent code.

Considerations when execute-only sections are present

Be aware of the following when *eXecute-Only* (XO) sections are present:

- You can mix XO and non-XO sections in the same execution region. In this case, the XO section loses its XO property and results in the output of a RO section.
- If an input file has one or more XO sections then the linker generates a separate XO execution region if the XO and RO sections are in distinct regions. In the final image, the XO execution region immediately precedes the RO execution region, unless otherwise specified by a scatter file or the `--xo_base` option.

  The linker automatically fabricates a separate ER_XO execution region for XO sections when all the following are true:

  - You do not specify the `--xo_base` option or a scatter file.
  - The input files contain at least one XO section.

Related information

Views of the image at each link stage on page 570
Methods of specifying an image memory map with the linker on page 576
Section placement with the linker on page 590

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.3.1.3 Load view and execution view of an image

Image regions are placed in the system memory map at load time. The location of the regions in memory might change during execution.

Before you can execute the image, you might have to move some of its regions to their execution addresses and create the ZI output sections. For example, initialized RW data might have to be copied from its load address in ROM to its execution address in RAM.

The memory map of an image has the following distinct views:

Load view

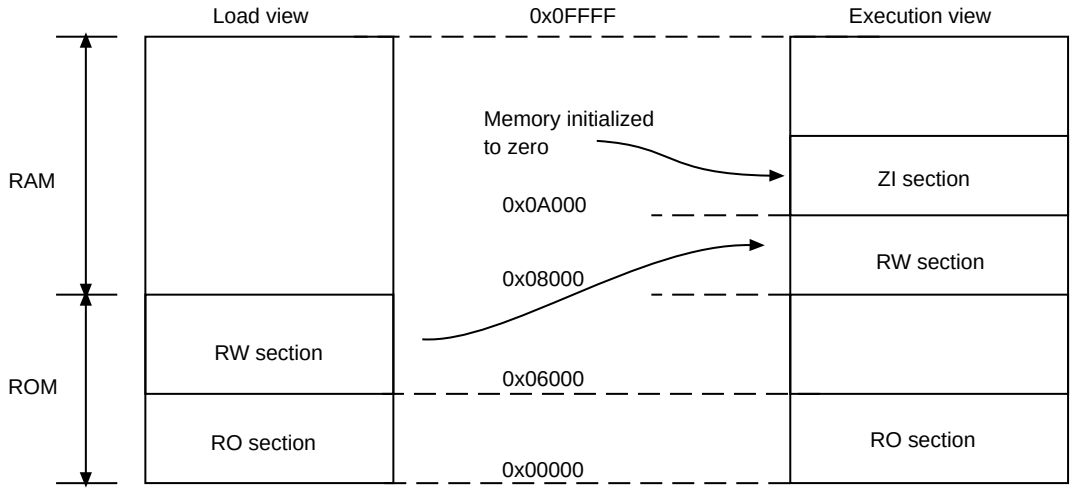
Describes each image region and section in terms of the address where it is located when the image is loaded into memory, that is, the location before image execution starts.

Execution view

Describes each image region and section in terms of the address where it is located during image execution.

The following figure shows these views for an image without an *eXecute-Only* (XO) section:

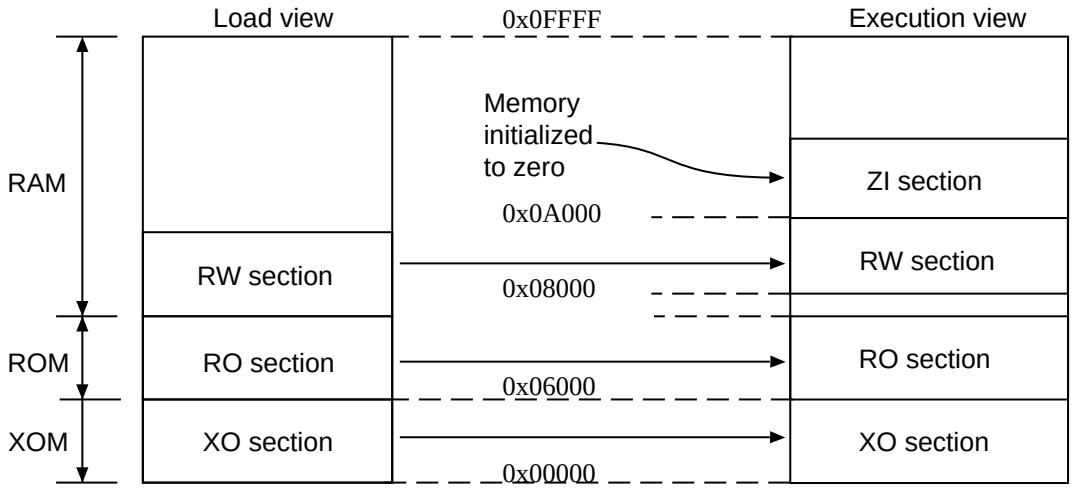
Figure 3-2: Load and execution memory maps for an image without an XO section



The following figure shows load and execution views for an image with an XO section:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 3-3: Load and execution memory maps for an image with an XO section



Note

XO memory is supported only for Arm®v6-M, Armv7-M, and Armv8-M architectures. However, XO is not supported on Armv6-M for any form of position independent code.

The following table compares the load and execution views:

Table 3-6: Comparing load and execution views

Load	Description	Execution	Description
Load address	The address where a section or region is loaded into memory before the image containing it starts executing. The load address of a section or a non-root region can differ from its execution address.	Execution address	The address where a section or region is located while the image containing it is being executed.
Load region	A load region describes the layout of a contiguous chunk of memory in load address space.	Execution region	An execution region describes the layout of a contiguous chunk of memory in execution address space.

Related information

Views of the image at each link stage on page 570
Methods of specifying an image memory map with the linker on page 576
Section placement with the linker on page 590
Input sections, output sections, regions, and program segments on page 572

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.3.1.4 Methods of specifying an image memory map with the linker

An image can consist of any number of regions and output sections. Regions can have different load and execution addresses.

When constructing the memory map of an image, `armlink` must have information about:

- How input sections are grouped into output sections and regions.
- Where regions are to be located in the memory map.

Depending on the complexity of the memory map of the image, there are two ways to pass this information to `armlink`:

Command-line options for simple memory map descriptions

You can use the following options for simple cases where an image has only one or two load regions and up to three execution regions:

- `--first`.
- `--last`.
- `--ro_base`.
- `--rosplit`.
- `--rw_base`.
- `--split`.
- `--xo_base`.
- `--zi_base`.

These options provide a simplified notation that gives the same settings as a scatter-loading description for a simple image. However, no limit checking for regions is available when using these options.

Scatter file for complex memory map descriptions

A scatter file is a textual description of the memory layout and code and data placement. It is used for more complex cases where you require complete control over the grouping and placement of image components. To use a scatter file, specify `--scatter=<filename>` at the command-line.

Note

You cannot use `--scatter` with the other memory map related command-line options.

Comparison of scatter file and equivalent command-line options

```
LR1 0x0000 0x20000
{
    ER_RO 0x0 0x2000           // --ro_base=0x0
    {
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
        init.o (INIT, +FIRST)  // --first=init.o(init)
        *(+RO)
    }

    ER_RW 0x400000             // --rw_base=0x400000
    {
        *(+RW)
    }

    ER_ZI 0x405000             // --zi_base=0x405000
    {
        *(+ZI)
    }
}

LR_XO 0x8000 0x4000
{
    ER_XO 0x8000               // --xo_base=0x8000
    {
        *(XO)
    }
}
```

Note

If XO sections are present, a separate load and execution region is created only when you specify `--xo_base`. If you do not specify `--xo_base`, then the ER_XO region is placed in the LR1 region at the address specified by `--ro_base`. The ER_RO region is then placed immediately after the ER_XO region.

Related information

Load view and execution view of an image on page 574
Simple images on page 581
The structure of an Arm ELF image on page 570
Input sections, output sections, regions, and program segments on page 572
--first=section_id on page 467
--last=section_id on page 489
--ro_base=address on page 525
--ropi on page 526
--rosplit on page 527
--rw_base=address on page 527
--rwpi on page 528
--scatter=filename on page 529
--split on page 538
--xo_base=address on page 560
--zi_base=address on page 562

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.3.1.5 Image entry points

An entry point in an image is the location that is loaded into the PC. It is the location where program execution starts. Although there can be more than one entry point in an image, you can specify only one when linking.

Not every ELF file has to have an entry point. Multiple entry points in a single ELF file are not permitted.


Note

For embedded programs targeted at a Cortex®-M based processor, the program starts at the location that is loaded into the PC from the Reset vector. Typically, the Reset vector points to the CMSIS `Reset_Handler` function.

Types of entry point

There are two distinct types of entry point:

Initial entry point

The initial entry point for an image is a single value that is stored in the ELF header file. It is the entry point where execution begins. For programs loaded into RAM by an operating system or boot loader, the loader starts the image execution by transferring control to the initial entry point in the image.

You can select one of many possible entry points for an image. An image can have only one initial entry point. The initial entry point can be, but is not required to be, one of the entry points set by the `ENTRY` directive for `armasm` legacy assembler.

Typically, the initial entry point is identified by `__main`. In C or C++, this initial entry point is the `main()` function. In GNU syntax assembler, you can identify this initial entry point with `.global main`. When `armlink` detects this initial entry point, it automatically includes the `__main` startup code from the C library, and sets that as the initial entry point.

For C and C++ code, the `__main()` function in the C library is also an entry point, and is typically the initial entry point for an image.

Other entry points

Entry points other than the initial entry point can be identified in code as follows:

Entry points in C and C++ code

If you have additional entry points in C or C++ code that are not reachable from the initial entry point, then you must use the `armlink` option `--keep` to identify those entry points.

Entry points in GNU syntax assembler

If you have additional entry points in GNU syntax assembler code that are not reachable from the initial entry point, then you must use the `armlink` option `--keep` to identify those entry points.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Entry points in armasm legacy assembler

For `armasm` legacy assembler code, you create entry points in objects with the `ENTRY` directive in an assembler file. In embedded systems, typical use of this directive is to mark code that is entered through the processor exception vectors, such as RESET, IRQ, and FIQ.

The directive marks the output code section with an `ENTRY` keyword that instructs the linker not to remove the section when it performs unused section elimination.

Alternatively, you could use the `armlink` option `--keep` to identify additional entry points in `armasm` legacy assembler code. Although Arm recommends writing new assembly code in GNU syntax assembly, if you have to write new `armasm` legacy assembler code, then use `--keep` to avoid any future migration problems.


Note

The `--keep` option prevents `armlink` from removing the sections for the related entry points during unused section elimination.

If an embedded image is to be used by a loader, it must have a single initial entry point specified in the header. Use either the `armclang` option `-e` or the `armlink` command-line option `--entry` to select the initial entry point.

Why have multiple entry points?

An entry point in an object file informs the linker that the location is expected to be referenced by an external entity. Therefore, the linker knows not to remove the section for that entry point when performing unused section elimination.

If you have multiple entry points, and none of them are `__main`, then you must use the `armlink` option `--entry=<location>` to specify the initial entry point for the image.

An example where multiple entry points can occur is a program that has a section containing interrupt vectors and another section that contains code where execution is to begin. The location where execution starts can be identified by the `main()` function in C or C++, or some other location in assembler code. In this case, a scatter file maps address 0 as the location of the vector table and the entry point for the image needs to be where execution begins. However, you do not want the linker to remove the section for the vector table, even if there are no references to it. To ensure that the vector table is not removed, where `vectors` identifies the section for the vector table:

- For GNU syntax assembler, link with the `--keep=vectors` options.
- For `armasm` legacy assembler, either use the `ENTRY` directive or the `armlink` option `--keep=vectors`.

In both cases, if the intended initial entry point is not `__main`, also link with `--entry=<location>` to specify the location where execution is to begin.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

-e on page 48
--entry=location on page 460
--keep=section_id (armlink) on page 482
ENTRY directive on page 991

3.3.1.5.1 The initial entry point for an image

There can be only one initial entry point for an image. If the linker cannot identify an initial entry point, the linker outputs the warning `L6305W`.

The initial entry point must meet the following conditions:

- The image entry point must always lie within an execution region.
- The execution region must not overlay another execution region, and must be a root execution region. That is, where the load address is the same as the execution address.

For C or C++ code, the initial entry point is the `main()` function.

For assembly code, if you do not use the `armlink` option `--entry` to specify the initial entry point:

GNU syntax assembly

- If the input objects contain only one entry point, then
  - If an assembler object contains `main` and is also identified with `.global main`, you do not have to use the `armlink` option `--entry=<location>`. In this case, `armlink` automatically includes the `__main` startup code from the C library.
  - If an assembler object does not contain `main` or is not identified with `.global main`, you must use the `armlink` option `--entry=<location>` to specify the corresponding location as the initial entry point for the image.
- The linker generates an image that does not contain an initial entry point when either:
  - One or more entry points is specified with the `armlink` option `--keep`.
  - No entry point is specified using the `armlink` option `--keep`.

armasm legacy assembly

- If the input objects contain only one entry point set by the `ENTRY` directive, the linker uses that entry point as the initial entry point for the image.
- The linker generates an image that does not contain an initial entry point when either:
  - More than one entry point is specified using the `ENTRY` directive.
  - No entry point is specified using the `ENTRY` directive.

For embedded applications with ROM at address zero use `--entry=0`, or optionally `0xFFFF0000` for processors that are using high vectors.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

High vectors are not supported in AArch64 state.

Note

Some processors, such as Cortex®-M7, can boot from a different address in some configurations.

Related information

Root region and the initial entry point on page 651
--entry=location on page 460
ENTRY directive on page 991
List of the armlink error and warning messages

3.3.1.6 Restrictions on image structure

When an instruction accesses a memory address on an AArch64 target, the data must be within 4GB of the program counter.

For example, consider the following scatter file:

```
LOAD_REGION 0x0000000000 0x200000
{
    ROOT_REGION +0
    {
        *(Init, +FIRST)
        * (+RO)
        * (+RW, +ZI)
    }
    STACKHEAP 0x1FFFF0 EMPTY -0x18000
    {
    }
}

LOAD_REGION2 0x4000000000 0x200000
{
    ROOT_REGION2 +0
    {
        *(high_mem)
    }
}
```

`LOAD_REGION2` is 16GB away from `LOAD_REGION`, so data in `high_mem` is not accessible from code in `LOAD_REGION`. This results in a relocation out of range error at link time.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.3.2 Simple images

A simple image consists of a number of input sections of type RO, RW, XO, and ZI. The linker collates the input sections to form the RO, RW, XO, and ZI output sections.

3.3.2.1 Types of simple image

The types of simple image the linker can create depends on how the output sections are arranged within load and execution regions.

The types are:

Type 1

One region in load view, four contiguous regions in execution view. Use the `--ro_base` option to create this type of image.

Any XO sections are placed in an ER_XO region at the address specified by `--ro_base`, with the ER_RO region immediately following the ER_XO region.

Type 2

One region in load view, four non-contiguous regions in execution view. Use the `--ro_base` and `--rw_base` options to create this type of image.

Type 3

Two regions in load view, four non-contiguous regions in execution view. Use the `--ro_base`, `--rw_base`, and `--split` options to create this type of image.

For all the simple image types when `--xo_base` is not specified:

- If any XO sections are present, the first execution region contains the XO output section. The address specified by `--ro_base` is used as the base address of this output section.
- The second execution region contains the RO output section. This output section immediately follows an XO output.
- The third execution region contains the RW output section, if present.
- The fourth execution region contains the ZI output section, if present.

These execution regions are referred to as, XO, RO, RW, and ZI execution regions.

When you specify `--xo_base`, then XO sections are placed in a separate load and execution region.

However, you can also use the `--rosplit` option for a Type 3 image. This option splits the default load region into two RO output sections, one for code and one for data.

You can also use the `--zi_base` command-line option to specify the base address of a ZI execution region for Type 1 and Type 2 images. This option is ignored if you also use the `--split` command-line option that is required for Type 3 images.

You can also create simple images with scatter files.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Equivalent scatter-loading descriptions for simple images on page 698
Type 1 image structure, one load region and contiguous execution regions on page 583
Type 2 image structure, one load region and non-contiguous execution regions on page 585
Type 3 image structure, multiple load regions and non-contiguous execution regions on page 587
--ro_base=address on page 525
--rosplit on page 527
--rw_base=address on page 527
--scatter=filename on page 529
--split on page 538
--xo_base=address on page 560
--zi_base=address on page 562

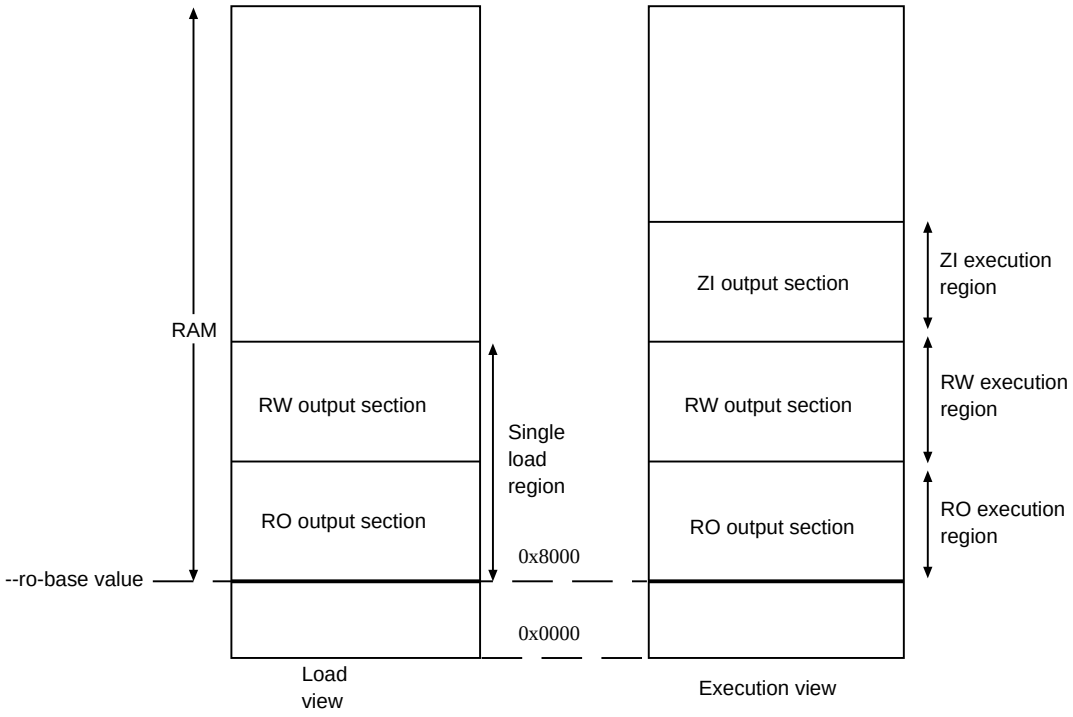
3.3.2.2 Type 1 image structure, one load region and contiguous execution regions

A Type 1 image consists of a single load region in the load view and three default execution regions, ER_RO, ER_RW, ER_ZI. These are placed contiguously in the memory map. An additional ER_XO execution region is created only if any input section is *eXecute-Only* (XO).

This approach is suitable for systems that load programs into RAM, for example, an OS bootloader or a desktop system. The following figure shows the load and execution view for a Type 1 image without XO code:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 3-4: Simple Type 1 image without execute-only code



Use the following command for images of this type:

```
armlink --cpu=8-A.32 --ro_base 0x8000
```

Note

`0x8000` is the default address, so you do not have to specify `--ro_base` for the example.

Load view

The single load region consists of the RO and RW output sections, placed consecutively. The RO and RW execution regions are both root regions. The ZI output section does not exist at load time. It is created before execution, using the output section description in the image file.

Execution view

The three execution regions containing the RO, RW, and ZI output sections are arranged contiguously. The execution addresses of the RO and RW regions are the same as their load addresses, so nothing has to be moved from its load address to its execution address. However, the ZI execution region that contains the ZI output section is created at run-time.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Use `armlink` option `--ro_base=<address>` to specify the load and execution address of the region containing the RO output. The default address is `0x8000`.

Use the `--zi_base` command-line option to specify the base address of a ZI execution region.

Load view for images containing execute-only regions

For images that contain XO sections, the XO output section is placed at the address that is specified by `--ro_base`. The RO and RW output sections are placed consecutively and immediately after the XO section.

Execution view for images containing execute-only regions

For images that contain XO sections, the XO execution region is placed at the address that is specified by `--ro_base`. The RO, RW, and ZI execution regions are placed contiguously and immediately after the XO execution region.



Note

XO memory is supported only for Arm®v6-M, Armv7-M, and Armv8-M architectures. However, XO is not supported on Armv6-M for any form of position independent code.

Related information

The structure of an Arm ELF image on page 570
Input sections, output sections, regions, and program segments on page 572
Load view and execution view of an image on page 574
--ro_base=address on page 525
--xo_base=address on page 560
--zi_base=address on page 562

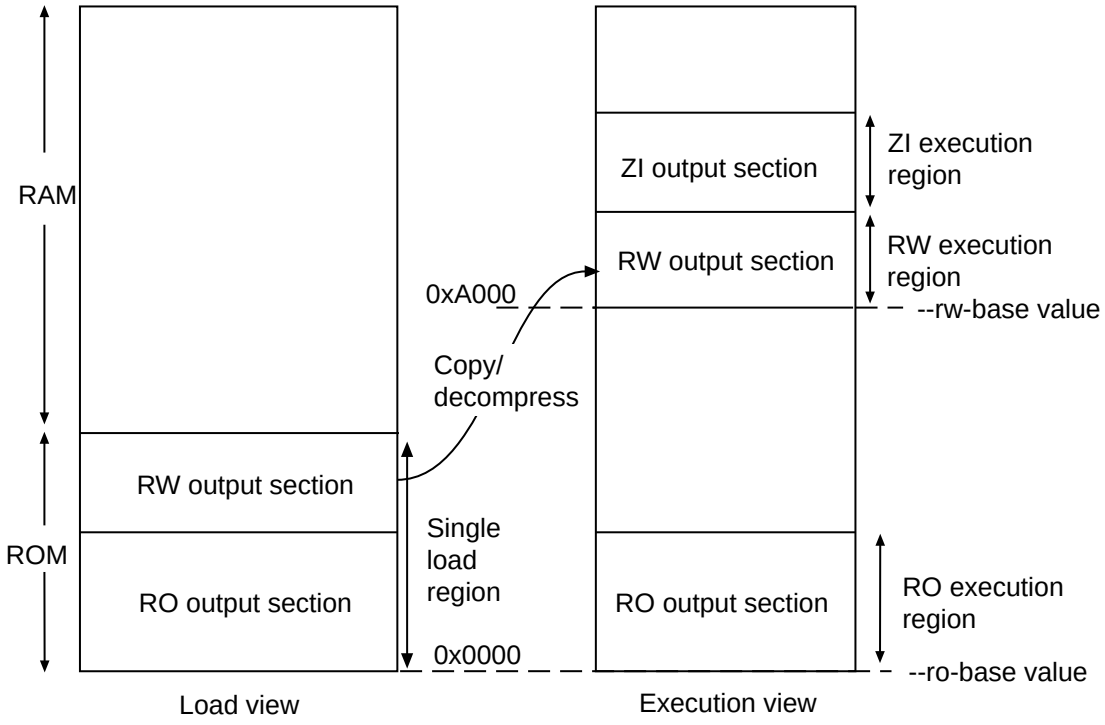
3.3.2.3 Type 2 image structure, one load region and non-contiguous execution regions

A Type 2 image consists of a single load region, and three execution regions in execution view. The RW execution region is not contiguous with the RO execution region.

This approach is used, for example, for ROM-based embedded systems, where RW data is copied from ROM to RAM at startup. The following figure shows the load and execution view for a Type 2 image without *eXecute-Only* (XO) code:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 3-5: Simple Type 2 image without execute-only code



Use the following command for images of this type:

```
armlink --cpu=8-A.32 --ro_base=0x0 --rw_base=0xA000
```

Note

XO memory is supported only for Arm®v6-M, Armv7-M, and Armv8-M architectures. However, XO is not supported on Armv6-M for any form of position independent code.

Load view

In the load view, the single load region consists of the RO and RW output sections placed consecutively, for example, in ROM. Here, the RO region is a root region, and the RW region is non-root. The ZI output section does not exist at load time. It is created at runtime.

Execution view

In the execution view, the first execution region contains the RO output section and the second execution region contains the RW and ZI output sections.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The execution address of the region containing the RO output section is the same as its load address, so the RO output section does not have to be moved. That is, it is a root region.

The execution address of the region containing the RW output section is different from its load address, so the RW output section is moved from its load address (from the single load region) to its execution address (into the second execution region). The ZI execution region, and its output section, is placed contiguously with the RW execution region.

Use `armlink` options `--ro_base=<address>` to specify the load and execution address for the RO output section, and `--rw_base=<address>` to specify the execution address of the RW output section. If you do not use the `--ro_base` option to specify the address, the default value of `0x8000` is used by `armlink`. For an embedded system, `0x0` is typical for the `--ro_base` value. If you do not use the `--rw_base` option to specify the address, the default is to place RW directly above RO (as in a Type 1 image).

Use the `--zi_base` command-line option to specify the base address of a ZI execution region.


Note

The execution region for the RW and ZI output sections cannot overlap any of the load regions.

Load view for images containing execute-only regions

For images that contain XO sections, the XO output section is placed at the address specified by `--ro_base`. The RO and RW output sections are placed consecutively and immediately after the XO section.

Execution view for images containing execute-only regions

For images that contain XO sections, the XO execution region is placed at the address specified by `--ro_base`. The RO execution region is placed contiguously and immediately after the XO execution region.

If you use `--xo_base <address>`, then the XO execution region is placed in a separate load region at the specified address.

Related information

The structure of an Arm ELF image on page 570
Input sections, output sections, regions, and program segments on page 572
Load view and execution view of an image on page 574
Type 1 image structure, one load region and contiguous execution regions on page 583
--ro_base=address on page 525
--rw_base=address on page 527
--xo_base=address on page 560
--zi_base=address on page 562

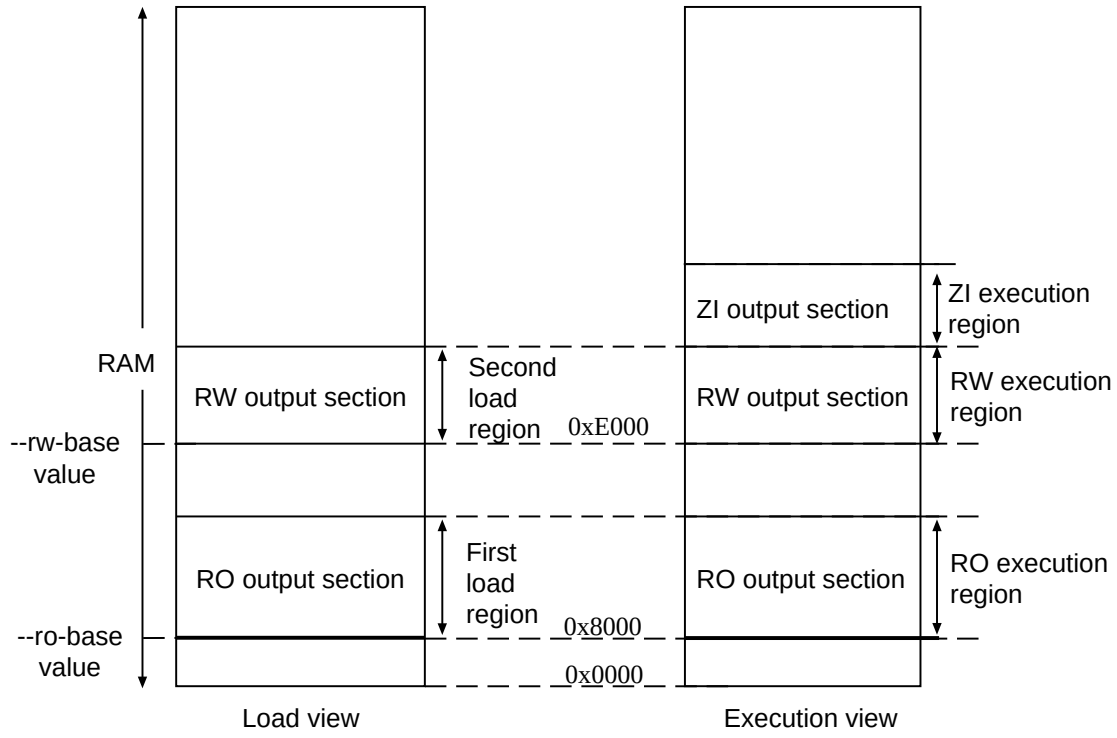
Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.3.2.4 Type 3 image structure, multiple load regions and non-contiguous execution regions

A Type 3 image is similar to a Type 2 image except that the single load region is split into multiple root load regions.

The following figure shows the load and execution view for a Type 3 image without *eXecute-Only* (XO) code:

Figure 3-6: Simple Type 3 image without execute-only code



Use the following command for images of this type:

```
armlink --cpu=8-A.32 --split --ro_base 0x8000 --rw_base 0xE000
```

Note

XO memory is supported only for Arm®v6-M, Armv7-M, and Armv8-M architectures. However, XO is not supported on Armv6-M for any form of position independent code.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Load view

In the load view, the first load region consists of the RO output section, and the second load region consists of the RW output section. The ZI output section does not exist at load time. It is created before execution, using the description of the output section contained in the image file.

Execution view

In the execution view, the first execution region contains the RO output section, the second execution region contains the RW output section, and the third execution region contains the ZI output section.

The execution address of the RO region is the same as its load address, so the contents of the RO output section do not have to be moved or copied from their load address to their execution address.

The execution address of the RW region is also the same as its load address, so the contents of the RW output section are not moved from their load address to their execution address. However, the ZI output section is created at run-time and is placed contiguously with the RW region.

Specify the load and execution address using the following linker options:

`--ro_base=<address>`

Instructs `armlink` to set the load and execution address of the region containing the RO section at a four-byte aligned `<address>`, for example, the address of the first location in ROM. If you do not use the `--ro_base` option to specify the address, the default value of `0x8000` is used by `armlink`.

`--rw_base=<address>`

Instructs `armlink` to set the execution address of the region containing the RW output section at a four-byte aligned `<address>`. If this option is used with `--split`, this specifies both the load and execution addresses of the RW region, for example, a root region.

`--split`

Splits the default single load region, that contains both the RO and RW output sections, into two root load regions:

- One containing the RO output section.
- One containing the RW output section.

You can then place them separately using `--ro_base` and `--rw_base`.

Load view for images containing XO sections

For images that contain XO sections, the XO output section is placed at the address specified by `--ro_base`. The RO and RW output sections are placed consecutively and immediately after the XO section.

If you use `--split`, then the one load region contains the XO and RO output sections, and the other contains the RW output section.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Execution view for images containing XO sections

For images that contain XO sections, the XO execution region is placed at the address specified by `--ro_base`. The RO execution region is placed contiguously and immediately after the XO execution region.

If you specify `--split`, then the XO and RO execution regions are placed in the first load region, and the RW and ZI execution regions are placed in the second load region.

If you specify `--xo_base <address>`, then the XO execution region is placed at the specified address in a separate load region from the RO execution region.

Related information

The structure of an Arm ELF image on page 570
Input sections, output sections, regions, and program segments on page 572
Load view and execution view of an image on page 574
Type 2 image structure, one load region and non-contiguous execution regions on page 585
--ro_base=address on page 525
--rw_base=address on page 527
--xo_base=address on page 560
--split on page 538

3.3.3 Section placement with the linker

The linker places input sections in a specific order by default, but you can specify an alternative sorting order if required.

3.3.3.1 Default section placement

By default, the linker places input sections in a specific order within an execution region.

The sections are placed in the following order:

1. By attribute as follows:
   a. Read-only code.
   b. Read-only data.
   c. Read-write code.
   d. Read-write data.
   e. Zero-initialized data.
2. By input section name if they have the same attributes. Names are considered to be case-sensitive and are compared in alphabetical order using the ASCII collation sequence for characters.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3. By a tie-breaker if they have the same attributes and section names. By default, it is the order that `armlink` processes the section. You can override the tie-breaker and sorting by input section name with the `FIRST` or `LAST` input section attribute.


Note

The sorting order is unaffected by ordering of section selectors within execution regions.

These rules mean that the positions of input sections with identical attributes and names included from libraries depend on the order the linker processes objects. This can be difficult to predict when many libraries are present on the command line. The `--tiebreaker=<cmdline>` option uses a more predictable order based on the order the section appears on the command line.

The base address of each input section is determined by the sorting order defined by the linker, and is correctly aligned within the output section that contains it.

The linker produces one output section for each attribute present in the execution region:

- One execute-only (XO) section if the execution region contains only XO sections.
- One RO section if the execution region contains read-only code or data.
- One RW section if the execution region contains read-write code or data.
- One ZI section if the execution region contains zero-initialized data.


Note

If an attempt is made to place data in an XO only execution region, then the linker generates an error.

XO sections lose the XO property if mixed with RO code in the same Execution region.

The XO and RO output sections can be protected at run-time on systems that have memory management hardware. RO and XO sections can be placed in ROM or Flash.

Alternative sorting orders are available with the `--sort=<algorithm>` command-line option. The linker might change the `<algorithm>` to minimize the amount of veneers generated if no algorithm is chosen.


Note

XO memory is supported only for Arm®v6-M, Armv7-M, and Armv8-M architectures. However, XO is not supported on Armv6-M for any form of position independent code.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

The following scatter file shows how the linker places sections:

```
LoadRegion 0x8000
{
    ExecRegion1 0x0000 0x4000
    {
        *(sections)
        *(moresections)
    }
    ExecRegion2 0x4000 0x2000
    {
        *(evenmoresections)
    }
}
```

The order of execution regions within the load region is not altered by the linker.

Related information

Relationship between the default armclang-generated sections and scatter-loading input sections on page 592
Placement of unassigned sections on page 593

3.3.3.1.1 Relationship between the default armclang-generated sections and scatter-loading input sections

How the default sections that `armclang` generates relate to the sections in an image depends on the attributes of the sections. Without a scatter file, the scatter-loading mechanism maps the sections using default input section selectors. However, you can modify the mapping with a scatter file.

The following table shows the relationship between `armclang`-generated sections and scatter-loading input sections:

Table 3-7: Relationship between the default armclang-generated sections and input section selectors

Default section names that armclang generates	Input section selectors for scatter-loading	Corresponding execution regions for scatter-loading without a scatter file	Example
`.bss`	`.bss`, `ZI`, or `BSS`	`ER_ZI`	Zero-Initialized data `int b=0;` and uninitialized data `int a;`.
`.data`	`.data`, `RW`, `RW-DATA`, or `DATA`	`ER_RW`	Read/write data `int z=5;`.
`.rodata`	`.rodata`, `RO`, `RO-DATA`, or `CONST`	`ER_RO`	Read-only numerical constants `int const x=5`.
`.rodata.str1.1`	`.rodata.str1.1`, `RO`, `RO-DATA`, or `CONST`	`ER_RO`	String contained in `printf("const string");`.
`.text`	`.text`, `RO`, `RO-CODE`, or `CODE`	`ER_RO`	Source that translates to instructions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Default section names that armclang generates	Input section selectors for scatter-loading	Corresponding execution regions for scatter-loading without a scatter file	Example
`.text` with `SHF_ARM_NOREAD` flag	`XO`	`ER_XO`	Source code built with `-mexecute_only`.
`.llvm.lto` (LLVM bitcode)	`RO-CODE`, or `CODE`	`ER_RO`	Section containing bitcode generated by the link time optimizer.

Note

This table shows the default section names that `armclang` generates. You can create sections with different names using the `__attribute__((section("<name>")))` function and variable attribute, for example.

`armlink` prioritizes the most specific selector first, with no ambiguity allowed. The input section selector `.rodata*` also selects `.rodata.str1.1`. Specifying both `*(.rodata*)` and `*(.rodata.str1.*)` matches `*(.rodata.str1.*)` sections then any remaining RO data sections with `*(.rodata*)`. For more information, see How the linker resolves multiple matches when processing scatter files.

Example

The following example shows the placement of code and data, with default section names and user-specified section names:

```
int x1 = 5;                              // in .data.x1 (default)
int y1[100];                             // in .bss.y1 (default)
int const z1[3] = {1,2,3};               // in .rodata.z1 (default)

int x2 __attribute__((section("foo"))) = 5;       // in foo (data part of region)
int y2[100];                                      // in .bss.y2 (default)
int const z2[3] __attribute__((section("bar"))) = {1,2,3};      // in bar
char *s2 __attribute__((section("foo"))) = "abc"; // s2 in foo, "abc"
 in .rodata.str1.1

int x3 __attribute__((section("foo"))) = 5;       // in foo (data part of region)
int y3[100];                                      // in .bss.y3 (default)
int const z3[3] = {1,2,3};                        // in .rodata.z3 (default)
char *s3 __attribute__((section("foo"))) = "def"; // s3 in foo, "def"
 in .rodata.str1.1

int add1(int x) __attribute__((section("foo")));
int add1(int x)                          // in foo (code part of region)
{
    return x+1;
}
```

Related information

#pragma clang section on page 322
__attribute__((section("name"))) function attribute on page 276

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.3.3.1.2 Placement of unassigned sections

The linker might not be able to place some input sections in any execution region.

When the linker is unable to place some input sections it generates an error message. This might occur because your current scatter file does not permit all possible module select patterns and input section selectors.

How you fix this depends on the importance of placing these sections correctly:

- If the sections must be placed at specific locations, then modify your scatter file to include specific module selectors and input section selectors as required.
- If the placement of the unassigned sections is not important, you can use one or more `.ANY` module selectors with optional input section selectors.

3.3.3.2 Section placement with the FIRST and LAST attributes

You can make sure that a section is placed either first or last in its execution region. For example, you might want to make sure the section containing the vector table is placed first in the image.

To do this, use one of the following methods:

- If you are not using scatter-loading, use the `--first` and `--last` linker command-line options to place input sections.
- If you are using scatter-loading, use the attributes `FIRST` and `LAST` in the scatter file to mark the first and last input sections in an execution region if the placement order is important.



`FIRST` and `LAST` must not violate the basic attribute sorting order. For example, `FIRST RW` is placed after any read-only code or read-only data.

Related information

The structure of an Arm ELF image on page 570
Input sections, output sections, regions, and program segments on page 572
Load view and execution view of an image on page 574
The scatter-loading mechanism on page 643
Syntax of an input section description on page 730
--first=section_id on page 467
--last=section_id on page 489

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.3.3.3 Section alignment with the linker

The linker ensures each input section starts at an address that is a multiple of the input section alignment.

When input sections have been ordered and before the base addresses are fixed, `armlink` inserts padding, if required, to force each input section to start at an address that is a multiple of the input section alignment.

`armlink` supports strict conformance with the ELF specification with the default option `--no_legacyalign`. The linker faults the base address of a region if it is not aligned so padding might be inserted to ensure compliance. With `--no_legacyalign`, the region alignment is the maximum alignment of any input section contained by the region.

If you use the option `--legacyalign`, the linker permits ELF program headers and output sections to be aligned on a four-byte boundary regardless of the maximum alignment of the input sections. This enables `armlink` to minimize the amount of padding that it inserts into the image.



Note

The `--legacyalign` and `--no_legacyalign` options are deprecated. The default behavior of the linker is equivalent to that of linking with `--no_legacyalign`.

If you are using scatter-loading, you can increase the alignment of a load region or execution region with the `ALIGN` attribute. For example, you can change an execution region that is normally four-byte aligned to be eight-byte aligned. However, you cannot reduce the natural alignment. For example, you cannot force two-byte alignment on a region that is normally four-byte aligned.

Related information

Load region attributes on page 715
Alignment of regions to page boundaries on page 693
--legacyalign, --no_legacyalign on page 490
Example of aligning a base address in execution space but still tightly packed in load space on page 743
Execution region attributes on page 723

3.3.4 Linker support for creating demand-paged files

The linker provides features for you to create files that are memory mapped.

In operating systems that support virtual memory, an ELF file can be loaded by mapping the ELF files into the address space of the process loading the file. When a virtual address in a page that is mapped to the file is accessed, the operating system loads that page from disk. ELF files that are to be used this way must conform to a certain format.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Use the `--paged` command-line option to enable demand paging mode. This helps produce ELF files that can be demand paged efficiently.

The basic constraints for a demand-paged ELF file are:

- There is no difference between the load and execution address for any output section.
- All PT_LOAD Program Headers have a minimum alignment, `pt_align`, of the page size for the operating system.
- All PT_LOAD Program Headers have a file offset, `pt_offset`, that is congruent to the virtual address (`pt_addr`) modulo `pt_align`.

When you specify `--paged`:

- The operating system page size is controlled by the `--pagesize` command-line option.
- The linker attempts to place the ELF Header and Program Header in the first PT_LOAD program header, if space is available.

Example: Demand-paged scatter file

This example shows a demand-paged scatter file that aligns `LR2` to a page boundary:

```
LR1 GetPageSize() + SizeOfHeaders()
{
    ER_RO +0
    {
        *(+RO)
    }
}
LR2 AlignExpr(+0, GetPageSize())
{
    ER_RW +0
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
}
```

Related information

Alignment of regions to page boundaries on page 693
The scatter-loading mechanism on page 643
--scatter=filename on page 529
GetPageSize() function on page 742
--paged on page 514
--pagesize=pagesize on page 514
AlignExpr(expr, align) function on page 741
SizeOfHeaders() function on page 743

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.3.5 Linker reordering of execution regions containing T32 code

The linker reorders execution regions containing T32 code only if the size of the T32 code exceeds the branch range.

If the code size of an execution region exceeds the maximum branch range of a T32 instruction, then `armlink` reorders the input sections using a different sorting algorithm. This sorting algorithm attempts to minimize the amount of veneers generated.

The T32 branch instructions that can be veneered are always encoded as a pair of 16-bit instructions. Processors that support Thumb®-2 technology have a range of 16MB. Processors that do not support Thumb-2 technology have a range of 4MB.

To disable section reordering, use the `--no_largeregions` command-line option.

Related information

Linker-generated veneers on page 597
--largeregions, --no_largeregions on page 487

3.3.6 Linker-generated veneers

Veneers are small sections of code generated by the linker and inserted into your program.

3.3.6.1 What is a veneer?

A veneer extends the range of a branch by becoming the intermediate target of the branch instruction.

The range of a `BL` instruction depends on the architecture:

- For AArch32 state, the range is 32MB for A32 instructions, 16MB for 32-bit T32 instructions, and 4MB for 16-bit T32 instructions. A veneer extends the range of the branch by becoming the intermediate target of the branch instruction. The veneer then sets the PC to the destination address.

  This enables the veneer to branch anywhere in the 4GB address space. If the veneer is inserted between A32 and T32 code, the veneer also handles instruction set state change.
- For AArch64 state, the range is 128MB. A veneer extends the range of the branch by becoming the intermediate target of the branch instruction. The veneer then loads the destination address and branches to it.

  This enables the veneer to branch anywhere in the 16EB address space.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

There are no state-change veneers in AArch64 state.

The linker can generate the following veneer types depending on what is required:

- Inline veneers.
- Short branch veneers.
- Long branch veneers.

`armlink` creates one input section called `Veneer$$Code` for each veneer. A veneer is generated only if no other existing veneer can satisfy the requirements. If two input sections contain a long branch to the same destination, only one veneer is generated that is shared by both branch instructions. A veneer is only shared in this way if it can be reached by both sections.

Note

If *eXecute-Only* (XO) sections are present, only XO-compliant veneer code is created in XO regions.

Related information

Veneer sharing on page 598
Veneer types on page 599
Generation of position independent to absolute veneers on page 600
Reuse of veneers when scatter-loading on page 600

3.3.6.2 Veneer sharing

If multiple objects result in the same veneer being created, the linker creates a single instance of that veneer. The veneer is then shared by those objects.

You can use the command-line option `--no_veneershare` to specify that veneers are not shared. This assigns ownership of the created veneer section to the object that created the veneer and so enables you to select veneers from a particular object in a scatter file, for example:

```
LR 0x8000
{
    ER_ROOT +0
    {
        object1.o(Veneer$$Code)
    }
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Be aware that veneer sharing makes it impossible to assign an owning object. Using `--no_veneershare` provides a more consistent image layout. However, this comes at the cost of a significant increase in code size, because of the extra veneers generated by the linker.

Related information

What is a veneer? on page 597
The scatter-loading mechanism on page 643
Scatter File Syntax on page 711
--veneershare, --no_veneershare on page 557

3.3.6.3 Veneer types

Veneers have different capabilities and use different code pieces.

The linker selects the most appropriate, smallest, and fastest depending on the branching requirements:

- Inline veneer:
  - Performs only a state change.
  - The veneer must be inserted just before the target section to be in range.
  - An A32 to T32 interworking veneer has a range of 256 bytes so the function entry point must appear within 256 bytes of the veneer.
  - A T32 to A32 interworking veneer has a range of zero bytes so the function entry point must appear immediately after the veneer.
  - An inline veneer is always position independent.
- Short branch veneer:
  - An interworking T32 to A32 short branch veneer has a range of 32MB, the range for an A32 instruction. An A64 short branch veneer has a range of 128MB.
  - A short branch veneer is always position independent.
  - A Range Extension T32 to T32 short branch veneer for processors that support Thumb®-2 technology.
- Long branch veneer:
  - Can branch anywhere in the address space.
  - All long branch veneers are also interworking veneers.
  - There are different long branch veneers for absolute or *Position Independent Code* (PIC).

When you are using veneers be aware of the following:

- The inline veneer limitations mean that you cannot move inline veneers out of an execution region using a scatter file. Use the command-line option `--no_inlineveneer` to prevent the generation of inline veneers.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- All veneers cannot be collected into one input section because the resulting veneer input section might not be within range of other input sections. If the sections are not within addressing range, long branching is not possible.
- The linker generates position independent variants of the veneers automatically. However, because such veneers are larger than non position independent variants, the linker only does this where necessary, that is, where the source and destination execution regions are both position independent and are rigidly related.

To optimize the code size of veneers, `armlink` chooses the variant in the order of preference:

1. Inline veneer.
2. Short branch veneer.
3. Long veneer.

Related information

What is a veneer? on page 597
--max_veneer_passes=value on page 505
--inlineveneer, --no_inlineveneer on page 480

3.3.6.4 Generation of position independent to absolute veneers

Calling from *position independent* (PI) code to absolute code requires a veneer.

The normal call instruction encodes the address of the target as an offset from the calling address. When calling from PI code to absolute code the offset cannot be calculated at link time, so the linker must insert a long-branch veneer.

The generation of PI to absolute veneers can be controlled using the `--piveneer` option, that is set by default. When this option is turned off using `--no_piveneer`, the linker generates an error when a call from PI code to absolute code is detected.

Related information

What is a veneer? on page 597
--max_veneer_passes=value on page 505
--piveneer, --no_piveneer on page 516

3.3.6.5 Reuse of veneers when scatter-loading

The linker reuses veneers whenever possible, but there are some limitations on the reuse of veneers in protected load regions and overlaid execution regions.

A scatter file enables you to create regions that share the same area of RAM:

- If you use the `PROTECTED` attribute for a load region it prevents:
  - Overlapping of load regions.
  - Veneer sharing.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

    - String sharing with the `--merge` option.
- If you use the `AUTO_OVERLAY` attribute for a region, no other execution region can reuse a veneer placed in an overlay execution region.
- If you use the `OVERLAY` attribute for a region, no other execution region can reuse a veneer placed in an overlay execution region.

If it is not possible to reuse a veneer, new veneers are created instead. Unless you have instructed the linker to place veneers somewhere specific using scatter-loading, a veneer is usually placed in the execution region that contains the call requiring the veneer. However, in some situations the linker has to place the veneer in an adjacent execution region, either to maximize sharing opportunities or for a short branch veneer to reach its target.

Related information

What is a veneer? on page 597
Load region attributes on page 715
Inheritance rules for load region address attributes on page 717
Inheritance rules for the RELOC address attribute on page 719
Inheritance rules for execution region address attributes on page 727
Overlay support in Arm Compiler for Embedded 6

3.3.6.6 Generation of secure gateway veneers

`armlink` can generate secure gateway veneers for symbols that are present in a Secure image. It can also output symbols to a specified output import library, when necessary.

`armlink` generates a secure gateway veneer when it finds in the Secure image an entry function that has both symbols `__acle_se_<entry>` and `<entry>` pointing to the same offset in the same section.

The secure gateway veneer is a sequence of two instructions:

```
<entry>:
        sg
        b.w __acle_se_<entry>
```

The original symbol `<entry>` is changed to point to the `SG` instruction of the secure gateway veneer.

You can specify an input import library and output import library with the following command-line options:

- `--import_cmse_lib_in=<filename>`.
- `--import_cmse_lib_out=<filename>`.

Placement of secure gateway veneers is controlled by an input import library and by a scatter file selection. The linker can also output addresses of secure gateways to an output import library.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

The following example shows the generation of a secure gateway veneer:

Input code:

```
        .text
entry:
__acle_se_entry:
        [entry's code]
        BXNS lr
```

Output code produced by `armlink`:

```
        .text
__acle_se_entry:
        [entry's code]
        BXNS lr

        .section Veneer$$CMSE, "ax"
entry:
        SG
        B.W __acle_se_entry
```

Related information

Placement of CMSE veneer sections for a Secure image on page 686
--import_cmse_lib_in=filename on page 472
--import_cmse_lib_out=filename on page 473
Overview of building Secure and Non-secure images with the Armv8-M Security Extension

3.3.7 Command-line options used to control the generation of C++ exception tables

You can control the generation of C++ exception tables using command-line options.

By default, or if the option `--exceptions` is specified, the image can contain exception tables. Exception tables are discarded silently if no code throws an exception. However, if the option `--no_exceptions` is specified, the linker generates an error if any exceptions tables are present after unused sections have been eliminated.

You can use the `--no_exceptions` option to ensure that your code is exceptions free. The linker generates an error message to highlight that exceptions have been found and does not produce a final image.

However, you can use the `--no_exceptions` option with the `--diag_warning` option to downgrade the error message to a warning. The linker produces a final image but also generates a message to warn you that exceptions have been found.

Related information

--diag_warning=tag[,tag,...] (armlink) on page 453

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--exceptions, --no_exceptions on page 464
-fexceptions, -fno-exceptions on page 56

3.3.8 Weak references and definitions

Weak symbol references and definitions provide additional flexibility in the way the linker handles various functions and variables in a build. This flexibility can help you work on larger projects, where different parts of the application are developed at different times in the project timeframe.

When global symbols are marked with a weak attribute, they are given the `STB_WEAK` symbol binding instead of the `STB_GLOBAL` symbol binding, as defined by the ELF specification. See Symbol Table for more information.

The following sections explain how you might work with weak symbol bindings to make it easier for you to manage your projects. To begin, we describe what is meant by global references and definitions.

Global symbol references and definitions

Global references and definitions are sometimes also referred to as non-weak, strong, or hard references and definitions.

The following example, `foo.c`, is not complicated, but is worth briefly describing how global definitions and references work, before describing weak definitions and weak references.

```
// foo.c
void bar(void);   // function prototype declaration for bar()

void foo(void)    // global function definition for foo()
{
    bar();        // global function reference to bar()
}
```

In this example, if the linker cannot resolve global references to symbols like `bar()` from the content loaded so far, that is, from the ELF objects and symbol listings explicitly specified on the command-line, it attempts to do so by finding the symbol in a library:

- If it is unable to find a definition to match the reference, the linker reports an error:

  ```
  Error: L6218E: Undefined symbol bar (referred from foo.o).
  ```

- If such a reference is resolved, a section that is reachable from an entry point by at least one non-weak reference is marked as used. This mark ensures the section is not removed by the linker unused section elimination optimization.

Each global reference must be resolved by exactly one definition. If there are multiple definitions, the linker reports an error, for example:

```
Error: L6200E: Symbol bar Multiply Defined
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

See How the linker resolves references for more information.

Weak symbol definitions

You can use a weak definition to resolve any reference to that symbol in the same way as a normal definition. However, if another non-weak definition of that symbol exists in the build, the linker uses that definition instead of the weak definition, and does not produce an error because of multiply-defined symbols.

You can mark a function or variable definition as weak in a source file, such as `foo.c`.

```
// foo.c
void foo(void) __attribute__ ((weak));   // weak function prototype declaration for
 foo()

volatile int a; // global variable declaration

void foo(void)  // default/placeholder function definition for foo()
{
    while(1);
}
```

If you read the output of the resulting ELF object with an ELF reader like `fromelf`, you notice that the `foo()` function in this example is given a weak (`Wk`) binding:

```
** Section #12 '.symtab' (SHT_SYMTAB)
    Size   : 112 bytes (alignment 4)
    String table #1 '.strtab'
    Last local symbol no. 4

    Symbol table .symtab (6 symbols, 4 local)

      #  Symbol Name                 Value      Bind  Sec  Type  Vis  Size
    ==========================================================================

      1  foo.c                       0x00000000   Lc  Abs  File  De
      2  [Anonymous Symbol]          0x00000000   Lc    3  Sect  De
      3  $t.0                        0x00000000   Lc    3   --   De
      4  [Anonymous Symbol]          0x00000000   Lc   10  Sect  De
      5  foo                         0x00000001   Wk    3  Code  Hi   0x4
      6  a                           0x00000000   Gb    6  Data  Hi   0x4
```

Create a new file `new_foo.c`:

```
// new_foo.c
void foo(void)
{
    // full implementation of foo() to replace 'weak' version
}
```

If you read the output of the resulting ELF object with an ELF reader like `fromelf`, you notice that the `foo()` function from `new_foo.c` is given a global (`Gb`) binding.

Weak function definitions are commonly used in real applications as default or placeholder exception handler routines. For example, CMSIS device startup code typically contains weak

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

function definitions for its exception handler routines. Therefore, there is a default behavior for things like fault handling, so you have something to start with.

```
// CMSIS startup code HardFault handler function declared as weak
void HardFault_Handler (void) __attribute__ ((weak));
```

Later on in the project timeframe, you are likely to provide your own full set of exception handling functions. You can create a function with exactly the same name as the weak exception handler functions. Then, rather than the linker issuing the `Multiply Defined` error message, it selects the strong, non-weak, version instead that contains your full implementation, now that it is ready.

For more information, see startup_ARMCM33.c

Weak references

Weak symbol references can be used in a similar way to weak symbol definitions. They can act as placeholders until the actual definition becomes available. This placeholder allows you the option to begin testing a particular part of the application, for example, at an earlier stage in the project, without having to wait for a new module that contains the definition to become available.

```
// foo.c
void bar(void) __attribute__ ((weak));   // weak function prototype declaration for
 bar()

void foo(void)    // global function definition for foo()
{
    bar();        // global function reference bar() with weak binding
}
```

This example contains a weak symbol reference to the function `bar()`. An unresolved weak function call is replaced with either:

- A no-operation instruction, `NOP`.
- A branch with link instruction, `BL`, to the following instruction. That is, the function call does not happen.

When the module that contains the definition for `bar()` is finally ready, it can be linked in instead.

Weak symbols in libraries

As stated in the Symbol Table section of the ELF specification:

"The link editor does not extract archive members to resolve undefined weak symbols."

Therefore, if there is a weak reference to a function contained in an object that is archived inside a library, the symbol is not necessarily pulled in by the linker to resolve a weak reference, unless the definition has already been loaded for one of the following reasons:

- The symbol has a non-weak reference from somewhere else in the code.
- The symbol definition exists in the same ELF section as a symbol definition that is included for any of these reasons. By default, function definitions are allocated their own sections because the `armclang` option `-ffunction-sections` is on by default. Therefore, unless you specify `-fno-`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`function-sections` or explicitly place multiple functions in the same section, this reason for the weak definition being loaded is less likely.

- The symbol definition is in a section that has been specified using `--keep`, or contains an entry point. See Image entry points for more information.
- The symbol definition is in an object file included in the link and the `--no_remove` option is used. The object file is not referenced from a library unless that object file within the library is explicitly included on the linker command-line.

Multiply-defined weak definitions

The linker issues the following warning to inform you that it made a choice from multiple available weak definitions:

```
L6439W: Multiply defined Global Symbol <sym> defined in <objname>(<secname>)
 rejected in favour of Symbol defined in <selobj>(<selsec>).
```

You can suppress this warning with the `--no_muldefweak` option.

Related information

--keep=section_id (armlink) on page 482
--muldefweak, --no_muldefweak on page 508
--remove, --no_remove on page 523
How the linker performs library searching, selection, and scanning on page 606
How the linker resolves references on page 609
-ffunction-sections, -fno-function-sections on page 67
__attribute__((weak)) function attribute on page 286
__attribute__((weak)) variable attribute on page 303
Symbol Table
List of the armlink error and warning messages

3.3.9 How the linker performs library searching, selection, and scanning

The linker always searches user libraries before the Arm libraries.

If you specify the `--no_scanlib` command-line option, the linker does not search for the default Arm libraries and uses only those libraries that are specified in the input file list to resolve references.

The linker creates an internal list of libraries as follows:

1. Any libraries explicitly specified in the input file list are added to the list.
2. The user-specified search path is examined to identify Arm standard libraries to satisfy requests embedded in the input objects.

   The best-suited library variants are chosen from the searched directories and their subdirectories. Libraries supplied by Arm have multiple variants that are named according to the attributes of their members.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Be aware of the following differences between the way the linker adds object files to the image and the way it adds libraries to the image:

- Each object file in the input list is added to the output image unconditionally, whether or not anything refers to it. At least one object must be specified.
- A member from a library is included in the output only if:
  - An object file or an already-included library member makes a non-weak reference to it.
  - The linker is explicitly instructed to add it.


Note

If a library member is explicitly requested in the input file list, the member is loaded even if it does not resolve any current references. In this case, an explicitly requested member is treated as if it is an ordinary object.

Unresolved references to weak symbols do not cause library members to be loaded.

Related information

How the linker searches for the Arm standard libraries on page 607

3.3.10 How the linker searches for the Arm standard libraries

The linker searches for the Arm standard libraries using information specified on the command-line, or by examining environment variables.

By default, the linker searches for the Arm standard libraries in `../lib`, relative to the location of the `armlink` executable. Use the `--libpath` command-line option to specify a different location.

The `--libpath` command-line option

Use the `--libpath` command-line option with a comma-separated list of parent directories. This list must end with the parent directory of the Arm library directories `armlib`, `cpplib`, and `libcxx`.

The sequential nature of the search ensures that `armlink` chooses the library that appears earlier in the list if two or more libraries define the same symbol.

Library search order

The linker searches for libraries in the following order:

1. At the location specified with the command-line option `--libpath`.
2. In `../lib`, relative to the location of the `armlink` executable.

How the linker selects Arm library variants

The Arm® Compiler for Embedded toolchain includes a number of variants of each of the libraries, that are built using different build options. For example, architecture versions, endianness, and instruction set. The variant of the Arm library is coded into the library name. The linker must select the best-suited variant from each of the directories identified during the library search.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The linker accumulates the attributes of each input object and then selects the library variant best suited to those attributes. If more than one of the selected libraries are equally suited, the linker retains the first library selected and rejects all others.

The `--no_scanlib` option prevents the linker from searching the directories for the Arm standard libraries.

Related information

--libpath=pathlist on page 491
How the linker performs library searching, selection, and scanning on page 606
C and C++ library naming conventions
The C and C++ libraries
Toolchain environment variables

3.3.11 Specifying user libraries when linking

You can specify your own libraries when linking.

Procedure

To specify user libraries, either:

- Include them with path information explicitly in the input file list.
- Add the `--userlibpath` option to the `armlink` command line with a comma-separated list of directories, and then specify the names of the libraries as input files.

You can use the `--library=<name>` option to specify static libraries, `lib<name>.a`.

If you do not specify a full path name to a library on the command line, the linker tries to locate the library in the directories specified by the `--userlibpath` option. For example, if the directory `/mylib` contains `my_lib.a` and `other_lib.a`, add `/mylib/my_lib.a` to the input file list with the command:

```
armlink --userlibpath /mylib my_lib.a *.o
```

If you add a particular member from a library this does not add the library to the list of searchable libraries used by the linker. To load a specific member and add the library to the list of searchable libraries include the library `<filename>` on its own as well as specifying `<library>(<member>)`. For example, to load `strcmp.o` and place `mystring.lib` on the searchable library list add the following to the input file list:

```
mystring.lib(strcmp.o) mystring.lib
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

Any search paths used for the Arm standard libraries specified by the linker command-line option `--libpath` are not searched for user libraries.

Related information

How the linker searches for the Arm standard libraries on page 607
--libpath=pathlist on page 491
--userlibpath=pathlist on page 554
The C and C++ libraries
Toolchain environment variables

3.3.12 How the linker resolves references

When the linker has constructed the list of libraries, it repeatedly scans each library in the list to resolve references.

`armlink` maintains two separate lists of files. The lists are scanned in the following order to resolve all dependencies:

1. The list of user files and libraries that have been loaded.
2. List of Arm standard libraries found in a directory relative to the `armlink` executable, or the directories specified by `--libpath`.

Each list is scanned using the following process:

1. Scan each of the libraries to load the required members:
   a. For each currently unsatisfied non-weak reference, search sequentially through the list of libraries for a matching definition. The first definition found is marked for processing in the next step.

      The sequential nature of the search ensures that the linker chooses the library that appears earlier in the list if two or more libraries define the same symbol. This enables you to override function definitions from other libraries, for example, the Arm C libraries, by adding your libraries to the input file list. However you must be careful to consistently override all the symbols in a library member. If you do not, you risk the objects from both libraries being loaded when there is a reference to an overridden symbol and a reference to a symbol that was not overridden. This results in a multiple symbol definition error `L6200E` for each overridden symbol.
   b. Load the library members marked in the previous step. As each member is loaded it might satisfy some unresolved references, possibly including weak ones. Loading a library member might also create new unresolved weak and non-weak references.
   c. Repeat these stages until all non-weak references are either resolved or cannot be resolved by any library.
2. If any non-weak reference remains unsatisfied at the end of the scanning operation, generate an error message.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

How the linker performs library searching, selection, and scanning on page 606
How the linker searches for the Arm standard libraries on page 607
Specifying user libraries when linking on page 608
--libpath=pathlist on page 491
Toolchain environment variables
List of the armlink error and warning messages

3.3.13 The strict family of linker options

The linker provides options to overcome the limitations of the standard linker checks.

The strict options are not directly related to error severity. Usually, you add a strict option because the standard linker checks are not precise enough or are potentially noisy with legacy objects.

The strict options are:

- `--strict`.
- `--[no_]strict_flags`.
- `--[no_]strict_ph`.
- `--[no_]strict_relocations`.
- `--[no_]strict_symbols`.
- `--[no_]strict_visibility`.

Related information

--strict on page 540
--strict_relocations, --no_strict_relocations on page 543
--strict_symbols, --no_strict_symbols on page 543
--strict_visibility, --no_strict_visibility on page 544

3.4 Linker Optimization Features

Describes the optimization features available in the Arm linker, `armlink`.

3.4.1 Elimination of common section groups

The linker can detect multiple copies of section groups, and discard the additional copies.

Arm® Compiler for Embedded generates complete objects for linking. Therefore:

- If there are inline functions in C and C++ sources, each object contains the out-of-line copies of the inline functions that the object requires.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- If templates are used in C++ sources, each object contains the template functions that the object requires.

When these functions are declared in a common header file, the functions might be defined many times in separate objects that are subsequently linked together. To eliminate duplicates, the compiler compiles these functions into separate instances of common section groups.

It is possible that the separate instances of common section groups, are not identical. Some of the copies, for example, might be found in a library that has been built with different, but compatible, build options, different optimization, or debug options.

If the copies are not identical, `armlink` retains the best available variant of each common section group, based on the attributes of the input objects. `armlink` discards the rest.

If the copies are identical, `armlink` retains the first section group located.

You control this optimization with the following linker options:

- Use the `--bestdebug` option to use the largest common data (COMDAT) group (likely to give the best debug view).
- Use the `--no_bestdebug` option to use the smallest COMDAT group (likely to give the smallest code size). This is the default.

  The image changes if you compile all files containing a COMDAT group A with `-g`, even if you use `--no_bestdebug`.

Related information

Elimination of unused sections on page 611

3.4.2 Elimination of unused sections

Elimination of unused sections is the most significant optimization on image size that the linker performs.

Unused section elimination:

- Removes unreachable code and data from the final image.
- Is suppressed in cases that might result in the removal of all sections.

To control this optimization, use the `armlink` options `--remove`, `--no_remove`, `--first`, `--last`, and `--keep`.

Unused section elimination requires an entry point. Therefore, if no entry point is specified for an image, use the `armlink` option `--entry` to specify an entry point.

Use the `armlink` option `--info unused` to instruct the linker to generate a list of the unused sections that it eliminates.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

`armlink` reports `Error: L6218E: Undefined symbol <symbol>` even if unused section removal has removed the requirement for this symbol. This behavior is different from the GNU linker, `ld`.

An input section is retained in the final image when:

- It contains an entry point or an externally accessible symbol. For example, an entry function into the secure code for the Arm®v8-M Security Extension.
- It is an `SHT_INIT_ARRAY`, `SHT_FINI_ARRAY`, or `SHT_PREINIT_ARRAY` section.
- It is specified as the first or last input section, either by the `--first` or `--last` option or by a scatter-loading equivalent.
- It is marked as unremovable by the `--keep` option.
- It is referred to, directly or indirectly, by a non-weak reference from an input section retained in the image.
- Its name matches the name referred to by an input section symbol, and that symbol is referenced from a section that is retained in the image.

Note

Compilers usually collect functions and data together and emit one section for each category. The linker can only eliminate a section if it is entirely unused.

You can mark a function or variable in source code with the `__attribute__((used))` attribute. This attribute causes `armclang` to generate the symbol `__tagsym$$used.<num>` for each function or variable, where `<num>` is a counter to differentiate each symbol. Unused section elimination does not remove a section that contains `__tagsym$$used.<num>`.

You can also use the `armclang` option `-ffunction-sections` to instruct the compiler to generate one ELF section for each function in the source file.

Related information

Elimination of common section groups on page 610

3.4.3 Optimization with RW data compression

RW data areas typically contain a large number of repeated values, such as zeros, that makes them suitable for compression.

RW data compression is enabled by default to minimize ROM size.

The linker compresses the data. This data is then decompressed on the target at run time.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The Arm libraries contain some decompression algorithms and the linker chooses the optimal one to add to your image to decompress the data areas when the image is executed. You can override the algorithm chosen by the linker.


Note

Not supported for AArch64 state.

3.4.3.1 How the linker chooses a compressor

`armlink` gathers information about the content of data sections before choosing the most appropriate compression algorithm to generate the smallest image.

If compression is appropriate, `armlink` can only use one data compressor for all the compressible data sections in the image. Different compression algorithms might be tried on these sections to produce the best overall size. Compression is applied automatically if:

```
Compressed data size + Size of decompressor < Uncompressed data size
```

When a compressor has been chosen, `armlink` adds the decompressor to the code area of your image. If the final image does not contain any compressed data, no decompressor is added.

Related information

Options available to override the compression algorithm used by the linker on page 613
Optimization with RW data compression on page 612
How compression is applied on page 614
Considerations when working with RW data compression on page 615

3.4.3.2 Options available to override the compression algorithm used by the linker

The linker has options to disable compression or to specify a compression algorithm to be used.

You can override the compression algorithm used by the linker by either:

- Using the `--datacompressor off` option to turn off compression.
- Specifying a compression algorithm.

To specify a compression algorithm, use the number of the required compressor on the linker command line, for example:

```
armlink --datacompressor 2 ...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Use the command-line option `--datacompressor list` to get a list of compression algorithms available in the linker:

```
armlink --datacompressor list
...
Num     Compression algorithm
=========================================================
0       Run-length encoding
1       Run-length encoding, with LZ77 on small-repeats
2       Complex LZ77 compression
```

When choosing a compression algorithm be aware that:

- Compressor 0 performs well on data with large areas of zero-bytes but few nonzero bytes.
- Compressor 1 performs well on data where the nonzero bytes are repeating.
- Compressor 2 performs well on data that contains repeated values.

The linker prefers compressor 0 or 1 where the data contains mostly zero-bytes (>75%). Compressor 2 is chosen where the data contains few zero-bytes (<10%). If the image is made up only of A32 code, then A32 decompressors are used automatically. If the image contains any T32 code, T32 decompressors are used. If there is no clear preference, all compressors are tested to produce the best overall size.



Note

It is not possible to add your own compressors into the linker. The algorithms that are available, and how the linker chooses to use them, might change in the future.

Related information

Optimization with RW data compression on page 612
How compression is applied on page 614
How the linker chooses a compressor on page 613
--datacompressor=opt on page 448
Considerations when working with RW data compression on page 615

3.4.3.3 How compression is applied

The linker applies compression depending on the compression type specified, and might apply additional compression on repeated phrases.

Run-length compression encodes data as non-repeated bytes and repeated zero-bytes. Non-repeated bytes are output unchanged, followed by a count of zero-bytes.

Lempel-Ziv 1977 (LZ77) compression keeps track of the last n bytes of data seen. When a phrase is encountered that has already been seen, it outputs a pair of values corresponding to:

- The position of the phrase in the previously-seen buffer of data.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The length of the phrase.

Related information

--datacompressor=opt on page 448

3.4.3.4 Considerations when working with RW data compression

There are some considerations to be aware of when working with RW data compression.

When working with RW data compression:

- Use the linker option `--map` to see where compression has been applied to regions in your code.
- If there is a reference from a compressed region to a linker-defined symbol that uses a load address, the linker turns off RW compression.
- If you are using an Arm® processor with on-chip cache, enable the cache after decompression to avoid code coherency problems.

Compressed data sections are automatically decompressed at run time, providing `__main` is executed, using code from the Arm libraries. This code must be placed in a root region. This is best done using `InRoot$$Sections` in a scatter file.

If you are using a scatter file, you can specify that a load or execution region is not to be compressed by adding the `NOCOMPRESS` attribute.

Related information

Optimization with RW data compression on page 612
How the linker chooses a compressor on page 613
Options available to override the compression algorithm used by the linker on page 613
Load$$ execution region symbols on page 625
Scatter-loading Features on page 643
--map, --no_map on page 504
How compression is applied on page 614
Scatter File Syntax on page 711

3.4.4 Function inlining with the linker

The linker inlines functions depending on what options you specify and the content of the input files.

The linker can inline small functions in place of a branch instruction to that function. For the linker to be able to do this, the function (without the return instruction) must fit in the four bytes of the branch instruction.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Use the `--inline` and `--no_inline` command-line options to control branch inlining. However, `--no_inline` only turns off inlining for user-supplied objects. The linker still inlines functions from the Arm standard libraries by default.

If branch inlining optimization is enabled, the linker scans each function call in the image and then inlines as appropriate. When the linker finds a suitable function to inline, it replaces the function call with the instruction from the function that is being called.

The linker applies branch inlining optimization before any unused sections are eliminated so that inlined sections can also be removed if they are no longer called.

Note

- For Arm®v7-A, the linker can inline two 16-bit encoded Thumb® instructions in place of the 32-bit encoded Thumb `BL` instruction.
- For Armv8-A and Armv8-M, the linker can inline two 16-bit T32 instructions in place of the 32-bit T32 `BL` instruction.

Use the `--info=inline` command-line option to list all the inlined functions.

Note

The linker does not inline small functions in AArch64 state.

Related information

Factors that influence function inlining on page 616
Elimination of unused sections on page 611
--info=topic[,topic,...] (armlink) on page 474
--inline, --no_inline on page 479

3.4.5 Factors that influence function inlining

There are a number of factors that influence how the linker inlines functions.

The following factors influence the way functions are inlined:

- The linker handles only the simplest cases and does not inline any instructions that read or write to the PC because this depends on the location of the function.
- If your image contains both A32 and T32 code, functions that are called from the opposite state must be built for interworking. The linker can inline functions containing up to two 16-bit T32 instructions. However, an A32 calling function can only inline functions containing either a single 16-bit encoded T32 instruction or a 32-bit encoded T32 instruction. The action that the linker takes depends on the size of the function being called. The following table shows the state of both the calling function and the function being called:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 3-8: Inlining small functions

Calling function state	Called function state	Called function size
A32	A32	4 to 8 bytes
A32	T32	2 to 6 bytes
T32	T32	2 to 6 bytes

The linker can inline in different states if there is an equivalent instruction available. For example, if a T32 instruction is `adds r0, r0` then the linker can inline the equivalent A32 instruction. It is not possible to inline from A32 to T32 because there is less chance of T32 equivalent to an A32 instruction.

- For a function to be inlined, the last instruction of the function must be either:

```
mov pc, lr
```

or

```
bx lr
```

A function that consists only of a return sequence can be inlined as a NOP.

- A conditional A32 instruction can only be inlined if either:
  - The condition on the BL matches the condition on the instruction being inlined. For example, BLEQ can only inline an instruction with a matching condition like ADDEQ.
  - The BL instruction or the instruction to be inlined is unconditional. An unconditional A32 BL can inline any conditional or unconditional instruction that satisfies all the other criteria. An instruction that cannot be conditionally executed cannot be inlined if the BL instruction is conditional.
- A BL that is the last instruction of a T32 If-Then (IT) block cannot inline a 16-bit encoded T32 instruction or a 32-bit MRS, MSR, or CPS instruction. This is because the IT block changes the behavior of the instructions within its scope so inlining the instruction changes the behavior of the program.

Related information

About branches that optimize to a NOP on page 618
Conditional instructions
ADD
B
CPS, CPSID, CPSIE
IT
MOV, MOVS (register)
MRS
MSR (register)

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.4.6 About branches that optimize to a NOP

Although the linker can replace branches with a NOP, there might be some situations where you want to stop this happening.

By default, the linker replaces any branch with a relocation that resolves to the next instruction with a NOP instruction. This optimization can also be applied if the linker reorders tail calling sections.

However, there are cases where you might want to disable the option, for example, when performing verification or pipeline flushes.

To control this optimization, use the `--branchnop` and `--no_branchnop` command-line options.

Related information

Linker reordering of tail calling sections on page 618
--branchnop, --no_branchnop on page 434

3.4.7 Linker reordering of tail calling sections

There are some situations when you might want the linker to reorder tail calling sections.

A tail calling section is a section that contains a branch instruction at the end of the section. If the branch instruction has a relocation that targets a function at the start of another section, the linker can place the tail calling section immediately before the called section. The linker can then optimize the branch instruction at the end of the tail calling section to a NOP instruction.

To take advantage of this behavior, use the command-line option `--tailreorder` to move tail calling sections immediately before their target.

Use the `--info=tailreorder` command-line option to display information about any tail call optimizations performed by the linker.



Note

The linker does not reorder tail calling functions in AArch64 state.

Related information

About branches that optimize to a NOP on page 618
Restrictions on reordering of tail calling sections on page 619
Veneer types on page 599
--info=topic[,topic,...] (armlink) on page 474
--tailreorder, --no_tailreorder on page 549

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.4.8 Restrictions on reordering of tail calling sections

There are some restrictions on the reordering of tail calling sections.

The linker:

- Can only move one tail calling section for each tail call target. If there are multiple tail calls to a single section, the tail calling section with an identical section name is moved before the target. If no section name is found in the tail calling section that has a matching name, then the linker moves the first section it encounters.
- Cannot move a tail calling section out of its execution region.
- Does not move tail calling sections before inline veneers.

Related information

Linker reordering of tail calling sections on page 618

3.4.9 Linker merging of comment sections

If input files have any comment sections that are identical, then the linker can merge them.

If input object files have any `.comment` sections that are identical, then the linker merges them to produce the smallest `.comment` section while retaining all useful information.

The linker associates each input `.comment` section with the filename of the corresponding input object. If it merges identical `.comment` sections, then all the filenames that contain the common section are listed before the section contents, for example:

```
file1.o
file2.o
.comment section contents.
```

The linker merges these sections by default. To prevent the merging of identical `.comment` sections, use the `--no_filtercomment` command-line option.


Note

`armlink` does not preprocess comment sections from `armclang`. If you do not want to retain the information in a `.comment` section, then use the `fromelf` command with the `--strip=comment` option to strip this section from the image.

Related information

--comment_section, --no_comment_section on page 441
--filtercomment, --no_filtercomment on page 465
--strip=option[,option,...] on page 823

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.4.10 Merging identical constants

The linker can attempt to merge identical constants in objects targeted at AArch32 state. The objects must be produced with Arm® Compiler for Embedded 6. If you compile with the `armclang -ffunction-sections` option, the merge is more efficient. This option is the default.

About this task

The following procedure is an example that shows the merging feature.

Note

If you use a scatter file, any regions that are marked with the `OVERLAY` or `PROTECTED` attribute affect the behavior of the `armlink --merge_litpools` option.

Procedure

1. Create a C source file, `litpool.c`, containing the following code:

```
int f1() {
    return 0xdeadbeef;
}
int f2() {
    return 0xdeadbeef;
}
```

2. Compile the source with `-S` to create an assembly file:

```
armclang -c -S -target arm-arm-none-eabi -mcpu=cortex-m0 -ffunction-sections \
    litpool.c -o litpool.s
```

Note

`-ffunction-sections` is the default.

Because `0xdeadbeef` is a difficult constant to create using instructions, a literal pool is created, for example:

```
...
f1:
...
    ldr     r0, __arm_cp.0_0
    bx      lr
...
__arm_cp.0_0:
    .long    3735928559             @ 0xdeadbeef
...
    .fnend

...
    .code    16                     @ @f2
    .thumb_func
f2:
...
    ldr     r0, __arm_cp.1_0
    bx      lr
...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
__arm_cp.1_0:
    .long    3735928559             @ 0xdeadbeef
...
    .fnend
...
```

Note

There is one copy of the constant for each function, because `armclang` cannot share these constants between both functions.

3. Compile the source to create an object:

```
armclang -c -target arm-arm-none-eabi -mcpu=cortex-m0 litpool.c -o litpool.o
```

4. Link the object file using the `--merge_litpools` option:

```
armlink --cpu=Cortex-M0 --merge_litpools litpool.o -o litpool.axf
```

Note

`--merge_litpools` is the default.

5. Run `fromelf` to view the image structure:

```
fromelf -c -d -s -t -v -z litpool.axf
```

The following example shows the result of the merge:

```
...
    f1
        0x00008000:    4801        .H      LDR      r0,[pc,#4] ; [0x8008] =
 0xdeadbeef
        0x00008002:    4770        pG      BX       lr
    f2
        0x00008004:    4800        .H      LDR      r0,[pc,#0] ; [0x8008] =
 0xdeadbeef
        0x00008006:    4770        pG      BX       lr
    $d.4
    __arm_cp.1_0
        0x00008008:    deadbeef    ....    DCD    3735928559
...
```

Related information

--merge_litpools, --no_merge_litpools on page 508
-ffunction-sections, -fno-function-sections on page 67
Interaction of OVERLAY and PROTECTED attributes with armlink merge options

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.5 Accessing and Managing Symbols with armlink

The Arm linker, `armlink`, uses various symbols that the compiler and assembler generate. These symbols allow the linker to place code and data, and allow your programs to access various attributes of load and execution regions.

The following symbols are available:

- Mapping symbols identify various inline transitions.
- Linker-defined symbols that are reserved by Arm, and that you can access if required.
- Region-related symbols that the linker generates for each region in the image.
- Section-related symbols that the linker generates when it creates an image without scatter-loading. The linker generates different types of section-related symbols for output and input sections.
- Special patterns, `$Super$$` and `$Sub$$`, that you can use for situations where an existing symbol cannot be modified or recompiled.

3.5.1 About mapping symbols

Mapping symbols are generated by the compiler and assembler to identify various inline transitions.

For Arm®v7-A, inline transitions can be between:

- Code and data at literal pool boundaries.
- ARM code and Thumb code, such as ARM and Thumb interworking veneers.

For Armv8-A, inline transitions can be between:

- Code and data at literal pool boundaries.
- A32 code and T32 code, such as A32/T32 interworking veneers.

For Armv6-M, Armv7-M, and Armv8-M, inline transitions can be between code and data at literal pool boundaries.

The mapping symbols available for each architecture are:

Symbol	Description	Architecture
`$a`	Start of a sequence of ARM/A32 instructions.	All
`$t`	Start of a sequence of Thumb/T32 instructions.	All
`$t.x`	Start of a sequence of ThumbEE instructions.	Armv7-A
`$d`	Start of a sequence of data items, such as a literal pool.	All
`$x`	Start of A64 code.	Armv8-A

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`armlink` generates the `$d.realdata` mapping symbol to communicate to `fromelf` that the data is from a non-executable section. Therefore, the code and data sizes output by `fromelf -z` are the same as the output from `armlink --info sizes`, for example:

```
      Code (inc. data)   RO Data

         x          y          z
```

In this example, the `y` is marked with `$d`, and `RO Data` is marked with `$d.realdata`.



Note

Symbols beginning with the characters `$v` are mapping symbols related to VFP and might be output when building for a target with VFP. Avoid using symbols beginning with `$v` in your source code.

Be aware that modifying an executable image with the `fromelf --elf --strip=localsymbols` command removes all mapping symbols from the image.

Related information

--list_mapping_symbols, --no_list_mapping_symbols on page 496
--strict_symbols, --no_strict_symbols on page 543
Symbol naming rules on page 947
--strip=option[,option,...] on page 823
--text on page 826
ELF for the Arm Architecture

3.5.2 Linker-defined symbols

The linker defines some symbols that are reserved by Arm, and that you can access if required.

Symbols that contain the character sequence `$$`, and all other external names containing the sequence `$$`, are names reserved by Arm.

You can import these symbolic addresses and use them as relocatable addresses by your assembly language programs, or refer to them as `extern` symbols from your C or C++ source code.

Be aware that:

- Linker-defined symbols are only generated when your code references them.
- If *eXecute-Only* (XO) sections are present, linker-defined symbols are defined with the following constraints:
  - XO linker defined symbols cannot be defined with respect to an empty region or a region that has no XO sections.
  - XO linker defined symbols cannot be defined with respect to a region that contains only RO sections.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- RO linker defined symbols cannot be defined with respect to a region that contains only XO sections.


Note

XO memory is supported only for Arm®v6-M, Armv7-M, and Armv8-M architectures. However, XO is not supported on Armv6-M for any form of position independent code.

Related information

Linker defined symbols and scatter files on page 628
Load$$ execution region symbols on page 625
Methods of importing linker-defined symbols in C and C++ on page 629
Methods of importing linker-defined symbols in Arm assembly language on page 632

3.5.3 Region-related symbols

The linker generates the different types of region-related symbols for each region in the image.

The types are:

- `Image$$` and `Load$$` for each execution region.
- `Load$$LR$$` for each load region.

If you are using a scatter file, the linker generates these symbols for each region in the scatter file.

If you are not using scatter-loading, the linker generates the symbols for the default region names. That is, the region names are fixed and the same types of symbol are supplied.

Related information

Image$$ execution region symbols on page 624
Load$$ execution region symbols on page 625
Load$$LR$$ load region symbols on page 626
Region name values when not scatter-loading on page 627

3.5.3.1 Image$$ execution region symbols

The linker generates `Image$$` symbols for every execution region present in the image.

The following table shows the symbols that the linker generates for every execution region present in the image. All the symbols refer to execution addresses after the C library is initialized.

Table 3-10: Image$$ execution region symbols

Symbol	Description
`Image$$<region_name>$$Base`	Execution address of the region.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Symbol	Description
`Image$$<region_name>$$Length`	Execution region length in bytes excluding ZI length.
`Image$$<region_name>$$Limit`	Address of the byte beyond the end of the non-ZI part of the execution region.
`Image$$<region_name>$$RO$$Base`	Execution address of the RO output section in this region.
`Image$$<region_name>$$RO$$Length`	Length of the RO output section in bytes.
`Image$$<region_name>$$RO$$Limit`	Address of the byte beyond the end of the RO output section in the execution region.
`Image$$<region_name>$$RW$$Base`	Execution address of the RW output section in this region.
`Image$$<region_name>$$RW$$Length`	Length of the RW output section in bytes.
`Image$$<region_name>$$RW$$Limit`	Address of the byte beyond the end of the RW output section in the execution region.
`Image$$<region_name>$$XO$$Base`	Execution address of the XO output section in this region.
`Image$$<region_name>$$XO$$Length`	Length of the XO output section in bytes.
`Image$$<region_name>$$XO$$Limit`	Address of the byte beyond the end of the XO output section in the execution region.
`Image$$<region_name>$$ZI$$Base`	Execution address of the ZI output section in this region.
`Image$$<region_name>$$ZI$$Length`	Length of the ZI output section in bytes.
`Image$$<region_name>$$ZI$$Limit`	Address of the byte beyond the end of the ZI output section in the execution region.

Related information

Region-related symbols on page 624

3.5.3.2 Load$$ execution region symbols

The linker generates `Load$$` symbols for every execution region present in the image.



Note

`Load$$<region_name>` symbols apply only to execution regions. `Load$$LR$$<load_region_name>` symbols apply only to load regions.

The following table shows the symbols that the linker generates for every execution region present in the image. All the symbols refer to load addresses before the C library is initialized.

Table 3-11: Load$$ execution region symbols

Symbol	Description
`Load$$<region_name>$$Base`	Load address of the region.
`Load$$<region_name>$$Length`	Region length in bytes.
`Load$$<region_name>$$Limit`	Address of the byte beyond the end of the execution region.
`Load$$<region_name>$$RO$$Base`	Address of the RO output section in this execution region.
`Load$$<region_name>$$RO$$Length`	Length of the RO output section in bytes.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Symbol	Description
`Load$$<region_name>$$RO$$Limit`	Address of the byte beyond the end of the RO output section in the execution region.
`Load$$<region_name>$$RW$$Base`	Address of the RW output section in this execution region.
`Load$$<region_name>$$RW$$Length`	Length of the RW output section in bytes.
`Load$$<region_name>$$RW$$Limit`	Address of the byte beyond the end of the RW output section in the execution region.
`Load$$<region_name>$$XO$$Base`	Address of the XO output section in this execution region.
`Load$$<region_name>$$XO$$Length`	Length of the XO output section in bytes.
`Load$$<region_name>$$XO$$Limit`	Address of the byte beyond the end of the XO output section in the execution region.
`Load$$<region_name>$$ZI$$Base`	Load address of the ZI output section in this execution region.
`Load$$<region_name>$$ZI$$Length`	Load length of the ZI output section in bytes. The Load Length of ZI is zero unless `<region_name>` has the `ZEROPAD` scatter-loading keyword set.
`Load$$<region_name>$$ZI$$Limit`	Load address of the byte beyond the end of the ZI output section in the execution region.

All symbols in this table refer to load addresses before the C library is initialized. Be aware of the following:

- The symbols are absolute because section-relative symbols can only have execution addresses.
- The symbols take into account RW compression.
- References to linker-defined symbols from RW compressed execution regions must be to symbols that are resolvable before RW compression is applied.
- If the linker detects a relocation from an RW-compressed region to a linker-defined symbol that depends on RW compression, then the linker disables compression for that region.
- Any zero bytes written to the file are visible. Therefore, the Limit and Length values must take into account the zero bytes written into the file.

Related information

Region-related symbols on page 624
Methods of importing linker-defined symbols in C and C++ on page 629
Methods of importing linker-defined symbols in Arm assembly language on page 632
Region name values when not scatter-loading on page 627
Optimization with RW data compression on page 612
Image$$ execution region symbols on page 624
Load$$LR$$ load region symbols on page 626
Execution region attributes on page 723

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.5.3.3 Load$$LR$$ load region symbols

The linker generates `Load$$LR$$` symbols for every load region present in the image.

A `Load$$LR$$` load region can contain many execution regions, so there are no separate `$$RO` and `$$RW` components.


Note

`Load$$LR$$<load_region_name>` symbols apply only to load regions. `Load$$<region_name>` symbols apply only to execution regions.

The following table shows the symbols that the linker generates for every load region present in the image.

Table 3-12: Load$$LR$$ load region symbols

Symbol	Description
`Load$$LR$$<load_region_name>$$Base`	Address of the load region.
`Load$$LR$$<load_region_name>$$Length`	Length of the load region.
`Load$$LR$$<load_region_name>$$Limit`	Address of the byte beyond the end of the load region.

Related information

Region-related symbols on page 624
The structure of an Arm ELF image on page 570
Input sections, output sections, regions, and program segments on page 572
Load view and execution view of an image on page 574

3.5.3.4 Region name values when not scatter-loading

When scatter-loading is not used when linking, the linker uses default region name values.

If you are not using scatter-loading, the linker uses region name values of:

- `ER_XO`, for an execute-only execution region, if present.
- `ER_RO`, for the read-only execution region.
- `ER_RW`, for the read-write execution region.
- `ER_ZI`, for the zero-initialized execution region.

You can insert these names into the following symbols to obtain the required address:

- `Image$$` execution region symbols.
- `Load$$` execution region symbols.

For example, `Load$$ER_RO$$Base`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Region-related symbols on page 624
Image$$ execution region symbols on page 624
Load$$ execution region symbols on page 625
Section-related symbols on page 633

3.5.3.5 Linker defined symbols and scatter files

When you are using scatter-loading, the names from a scatter file are used in the linker defined symbols.

The scatter file:

- Names all the load and execution regions in the image, and provides their load and execution addresses. These names begin with `Load$$` for load regions and `Image$$` for execution regions.
- Defines both stack and heap. The linker also generates special stack and heap symbols.

Example: Define a symbol for the address of an execution region in a scatter file and access the address in C code

Create the file `main.c` containing:

```
#include <arm_acle.h>
#include <stdio.h>

extern uint32_t Image$$ER_RW_DATA$$Base;
extern uint32_t Image$$ER_RW_DATA$$Length;
extern uint32_t Image$$ER_RW_DATA$$RW$$Limit;

int main(void) {
    uint32_t rw_base_addr;
    uint32_t rw_end_addr;
    uint32_t rw_limit_addr;

    // Linker-defined symbols for the ER_RW_DATA
    // execution region.
    rw_base_addr = (uint32_t)&Image$$ER_RW_DATA$$Base;
    rw_end_addr = rw_base_addr + (uint32_t)&Image$$ER_RW_DATA$$Length;
    rw_limit_addr = (uint32_t)&Image$$ER_RW_DATA$$RW$$Limit;
    printf("ER_RW_DATA Base address:  0x%lX\n", (unsigned long)rw_base_addr);
    printf("ER_RW_DATA end address:   0x%lX\n", (unsigned long)rw_end_addr);
    printf("ER_RW_DATA limit address: 0x%lX\n", (unsigned long)rw_limit_addr);
}
```

Create the file `include_file.h` containing:

```
#define ER_RW_DATA_BASE_ADDR 0x400000
#define ER_RW_DATA_SIZE 0x100
```

Create the file `scatter.scat` containing:

```
#! armclang --target=arm-arm-none-eabi -mcpu=cortex-a8 -E -x c
#include "include_file.h"
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
LR 0x8000
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW_DATA
    ER_RW_DATA_BASE_ADDR
    ER_RW_DATA_SIZE
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
    ARM_LIB_STACKHEAP +0 EMPTY 0x8000
    {
    }
}
```

Compile with:

```
armclang -c --target=arm-arm-none-eabi -mcpu=cortex-a8 -o main.o main.c
```

Link with:

```
armlink --map --list=mappings.txt --scatter=scatter.scat -o main.axf main.o
```

The file `mappings.txt` shows the maximum size of the execution region to be that from the defined `ER_RW_DATA_SIZE`:

```
Execution Region ER_RW_DATA (Base: 0x00400000, Size: 0x00000010, Max: 0x00000100,
 ABSOLUTE)
```

If you load the image onto a suitable target and run it, the output shows:

```
ER_RW_DATA Base address:  0x400000
ER_RW_DATA end address:   0x400010
ER_RW_DATA limit address: 0x400010
```

Although we specified a maximum size of `0x100` for the `ER_RW_DATA`, the reported size using the linker-defined symbols is `0x10`.

Related information

Scatter-loading Features on page 643
--scatter=filename on page 529

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.5.3.6 Methods of importing linker-defined symbols in C and C++

You can import linker-defined symbols into your C or C++ source code. They are external symbols and you must take the address of them.

The only case where the `&` operator is not required is when the array declaration is used, for example `extern char symbol_name[];`.

Accessing a linker-defined symbol by casting its address to an integer type permits the compiler to assume the value of the symbol is non-null. However, compiler optimizations can result in the compiler generating code that results in unexpected run-time behavior.

Workarounds

A linker-defined symbol can be 0, for example, if an execution region containing zero-initialized memory, `ER_ZI`, is empty it would have a length of zero. In this case, use one of the following workarounds:

- Place the code that takes the address of the symbol in a separate source file.

  
  Note

  If you compile with *Link-Time Optimization* (LTO), `armclang` might generate code that results in unexpected run-time behavior.

- Use the `volatile` keyword.
- Use `memcpy` directly rather than a `for` loop that the compiler transforms into a `memcpy`.
- Define the `length` as `Load$$DATA_RAM$$RW$$Limit - Load$$DATA_RAM$$RW$$Base`.

Example: Equivalent of the built-in scatter file

Without a scatter file, `ER_RW` is the built-in name for RW. The following scatter file is a simplified version of the built-in scatter file:

```
LR1 0x8000 {
    ER_RO +0 {
        *(+RO)
    }
    ER_RW 0x10000000 {
        *(+RW)
    }
    ER_ZI +0 {
        *(+ZI)
    }
}
```

Example: Importing a linker-defined symbol using a volatile variable

Define the linker-defined symbol as a `volatile` variable:

```
extern unsigned long Image$$DATA_RAM$$RW$$Length;

volatile unsigned long DATA_RAM__RW_Size_volatile =
    ( unsigned long )&Image$$DATA_RAM$$RW$$Length;
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
unsigned long DATA_RAM__RW_Size = DATA_RAM__RW_Size_volatile;
```

The compiler does not make an assumption about the value of `DATA_RAM__RW_Size` because it considers the `volatile` variable might have changed between the write and the read.

Example: Importing a linker-defined symbol using separate source files

Example `use.c` file:

```
// Code to access the variable that contains the
// value of the linker-defined symbol
extern int er_zi_limit;
...
    // Compiler cannot assume that er_zi_limit is non 0
    if (er_zi_limit == 0) {
        return;
    }
...
```

Example `linker_defined_symbols.c` file:

```
// Declare the linker-defined symbol
extern size_t Image$$ER_ZI$$Length;
// Make a global variable with the address of the
// symbol cast to size_t
size_t er_zi_limit = (size_t)&Image$$ER_ZI$$Length;
```

Example: Importing symbols that define a ZI output section

This example passes the linker-defined symbols as parameters:

```
extern int Image$$ER_ZI$$Length;
extern char Image$$ER_ZI$$Base[];
memset(Image$$ER_ZI$$Base, 0, (size_t)&Image$$ER_ZI$$Length);
```

Example: Importing symbols for use in a for loop without a volatile variable

```
#include <stdint.h>

extern unsigned char Load$$ER_RW$$Base;
extern unsigned char Image$$ER_RW$$Base;
extern int Image$$ER_RW$$Length;

void copy_er_rw()
{
    char* srce = (char *)&Load$$ER_RW$$Base;
    char* dest = (char *)&Image$$ER_RW$$Base;
    unsigned int length = (uintptr_t)&Image$$ER_RW$$Length;

    // The loop optimizer of the compiler can assume that
    // the srce, dest, and length are non-null or non-0.
    // Therefore, the compiler can transform this for loop
    // into a do-while loop with at least one iteration.
    for(int i = 0; i < length ; i++)
        *dest++ = *srce++;
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Importing symbols for use in a for loop and with a volatile variable

If any of your symbols can have the value 0 then you can use `volatile`. For example:

```
#include <stdint.h>

extern unsigned char Load$$ER_RW$$Base;
extern unsigned char Image$$ER_RW$$Base;
extern int Image$$ER_RW$$Length;

void copy_er_rw()
{
    // Assert the ER_RW load address != 0
    char* srce = (char *)&Load$$ER_RW$$Base;

    // Assert the ER_RW execution address !=0
    char* dest = (char *)&Image$$ER_RW$$Base;

    // ER_RW length might be 0 if no RW
    volatile int length = (uintptr_t)&Image$$ER_RW$$Length;

    // The compiler cannot assume that the length is
    // not 0 because of volatile.
    for(int i = 0; i < length ; i++)
        *dest++ = *srce++;
}
```

Related information

Image$$ execution region symbols on page 624

3.5.3.7 Methods of importing linker-defined symbols in Arm assembly language

You can import linker-defined symbols into your Arm assembly code.

To import linker-defined symbols into your assembly language source code, use the `.global` directive.

32-bit applications

Create a 32-bit data word to hold the value of the symbol, for example:

```
        .global Image$$ER_ZI$$Limit
        ...
.zi_limit:
        .word Image$$ER_ZI$$Limit
```

To load the value into a register, such as r1, use the `LDR` instruction:

```
        ldr r1, .zi_limit
```

The `LDR` instruction must be able to reach the 32-bit data word. The accessible memory range varies between A64, A32, and T32, and the architecture you are using.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

64-bit applications

Create a 64-bit data word to hold the value of the symbol, for example:

```
        .global Image$$ER_ZI$$Limit
        ...
.zi_limit:
        .quad Image$$ER_ZI$$Limit
```

To load the value into a register, such as x1, use the `LDR` instruction:

```
        ldr x1, .zi_limit
```

The `LDR` instruction must be able to reach the 64-bit data word.

Related information

Image$$ execution region symbols on page 624
IMPORT and EXTERN directives on page 1010
A32 and T32 Instructions

3.5.4 Section-related symbols

Section-related symbols are symbols generated by the linker when it creates an image without scatter-loading. The linker generates different types of section-related symbols for output and input sections.

The types of symbols are:

- Image symbols, if you do not use scatter-loading to create a simple image. A simple image has up to four output sections (XO, RO, RW, and ZI) that produce the corresponding execution regions. See Image symbols for more information.
- Input section symbols, for every input section present in the image. See Input section symbols for more information.

The linker sorts sections within an execution region first by attribute RO, RW, or ZI, then by name. So, for example, all `.text` sections are placed in one contiguous block. A contiguous block of sections with the same attribute and name is known as a consolidated section .

3.5.4.1 Image symbols

Image symbols are generated by the linker when you do not use scatter-loading to create a simple image.

The following table shows the image symbols:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 3-13: Image symbols

Symbol	Section type	Description
`Image$$RO$$Base`	Output	Address of the start of the RO output section.
`Image$$RO$$Limit`	Output	Address of the first byte beyond the end of the RO output section.
`Image$$RW$$Base`	Output	Address of the start of the RW output section.
`Image$$RW$$Limit`	Output	Address of the byte beyond the end of the ZI output section. (The choice of the end of the ZI region rather than the end of the RW region is to maintain compatibility with legacy code.)
`Image$$ZI$$Base`	Output	Address of the start of the ZI output section.
`Image$$ZI$$Limit`	Output	Address of the byte beyond the end of the ZI output section.

Note

- We recommend that you use region-related symbols in preference to section-related symbols.
- The ZI output sections of an image are not created statically, but are automatically created dynamically at runtime.
- There are no load address symbols for RO, RW, and ZI output sections.

If you are using a scatter file, the image symbols are undefined. If your code accesses any of these symbols, you must treat them as a weak reference.

The standard implementation of `__user_setup_stackheap()` uses the value in `Image$$ZI$$Limit`. Therefore, if you are using a scatter file you must manually place the stack and heap. You can do this either:

- In a scatter file using one of the following methods:
  - Define separate stack and heap regions called `ARM_LIB_STACK` and `ARM_LIB_HEAP`.
  - Define a combined region containing both stack and heap called `ARM_LIB_STACKHEAP`.
- By reimplementing `__user_setup_stackheap()` to set the heap and stack boundaries.

Related information

Linker-defined symbols that are not defined when scatter-loading on page 645
Placing the stack and heap with a scatter file on page 645
Simple images on page 581
Weak references and definitions on page 603
__user_setup_stackheap()

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.5.4.2 Input section symbols

Input section symbols are generated by the linker for every input section present in the image.

The following table shows the input section symbols:

Table 3-14: Section-related symbols

Symbol	Section type	Description
`<SectionName>$$Base`	Input	Address of the start of the consolidated section called `<SectionName>`.
`<SectionName>$$Length`	Input	Length of the consolidated section called `<SectionName>` (in bytes).
`<SectionName>$$Limit`	Input	Address of the byte beyond the end of the consolidated section called `<SectionName>`.

If your code refers to the input-section symbols, it is assumed that you expect all the input sections in the image with the same name to be placed contiguously in the image memory map.

If your scatter file places input sections non-contiguously, the linker issues an error. This is because the use of the base and limit symbols over non-contiguous memory is ambiguous.

Related information

Scatter-loading Features on page 643
Input sections, output sections, regions, and program segments on page 572

3.5.5 Access symbols in another image

Use a *symbol definitions* (symdefs) file if you want one image to know the global symbol values of another image.

3.5.5.1 Creating a symdefs file

You can specify a *symbol definitions* (symdefs) file on the linker command-line.

About this task

You can use a symdefs file, for example, if you have one image that always resides in ROM and multiple images that are loaded into RAM. The images loaded into RAM can access global functions and data from the image located in ROM.

Procedure

Use the `armlink` option `--symdefs=<filename>` to generate a symdefs file.
The linker produces a symdefs file during a successful final link stage. It is not produced for partial linking or for unsuccessful final linking.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

If `<filename>` does not exist, the linker creates the file and adds entries for all the global symbols to that file. If `<filename>` exists, the linker uses the existing contents of `<filename>` to select the symbols that are output when it rewrites the file. This means that only the existing symbols in the filename are updated, and no new symbols (if any) are added at all. If you do not want this behavior, ensure that any existing symdefs file is deleted before the link step.

Related information

Outputting a subset of the global symbols on page 636
Symdefs file format on page 637
--symdefs=filename on page 546
Reading a symdefs file on page 636

3.5.5.2 Outputting a subset of the global symbols

You can use a symdefs file to output a subset of the global symbols to another application.

About this task

By default, all global symbols are written to the symdefs file. When a symdefs file exists, the linker uses its contents to restrict the output to a subset of the global symbols.

This example uses an application `image1` containing symbols that you want to expose to another application using a symdefs file.

Procedure

1. Specify `--symdefs=<filename>` when you are doing a final link for `image1`. The linker creates a symdefs file `<filename>`.
2. Open `<filename>` in a text editor, remove any symbol entries you do not want in the final list, and save the file.
3. Specify `--symdefs=<filename>` when you are doing a final link for `image1`.
   You can edit `<filename>` at any time to add comments and link `image1` again. For example, to update the symbol definitions to create `image1` after one or more objects have changed.

   You can use the symdefs file to link additional applications.

Related information

Creating a symdefs file on page 635
Symdefs file format on page 637
--symdefs=filename on page 546
Access symbols in another image on page 635

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.5.5.3 Reading a symdefs file

A *symbol definitions* (symdefs) file can be considered as an object file with symbol information but no code or data.

Procedure

To read a symdefs file, add it to your file list as you do for any object file. The linker reads the file and adds the symbols and their values to the output symbol table. The added symbols have `ABSOLUTE` and `GLOBAL` attributes.
If a partial link is being performed, the symbols are added to the output object symbol table. If a full link is being performed, the symbols are added to the image symbol table.

The linker generates error messages for invalid rows in the file. A row is invalid if:

- Any of the columns are missing.
- Any of the columns have invalid values.

The symbols extracted from a symdefs file are treated in exactly the same way as symbols extracted from an object symbol table. The same restrictions apply regarding multiple symbol definitions.



Note

The same function name or symbol name cannot be defined in both A32 code and in T32 code.

Related information

Symdefs file format on page 637

3.5.5.4 Symdefs file format

A *symbol definitions* (symdefs) file defines symbols and their values.

The file consists of:

Identification line

The identification line in a symdefs file comprises:

- An identifying string, `#<SYMDEFS>#`, which must be the first 11 characters in the file for the linker to recognize it as a symdefs file.
- Linker version information, in the format:

```
ARM Linker, <vvvvbbb>:
```

- Date and time of the most recent update of the symdefs file, in the format:

```
Last Updated: {day month date hh:mm:ss year}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

For example, for version 6.3, build 169:

```
#<SYMDEFS># ARM Linker, 6030169: Last Updated: Thu Jun 4 12:49:45 2015
```

The version and update information are not part of the identifying string.

Comments

You can insert comments manually with a text editor. Comments have the following properties:

- The first line must start with the special identifying comment `#<SYMDEFS>#`. This comment is inserted by the linker when the file is produced and must not be manually deleted.
- Any line where the first non-whitespace character is a semicolon (`;`) or hash (`#`) is a comment.
- A semicolon (`;`) or hash (`#`) after the first non-whitespace character does not start a comment.
- Blank lines are ignored and can be inserted to improve readability.

Symbol information

The symbol information is provided on a single line, and comprises:

Symbol value

The linker writes the absolute address of the symbol in fixed hexadecimal format, for example, `0x00008000`. If you edit the file, you can use either hexadecimal or decimal formats for the address value.

Type flag

A single letter to show symbol type:

X
A64 code (AArch64 only)

A
A32 code (AArch32 only)

T
T32 code (AArch32 only)

D
Data

N
Number.

Symbol name

The symbol name.

Example: A typical symdefs file format

```
#<SYMDEFS># ARM Linker, 6030169: Last Updated: <Date>

;value type name, this is an added comment
0x00008000 A __main
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
0x00008004 A __scatterload
0x000080E0 T main
0x0000814D T _main_arg
0x0000814D T __argv_alloc
0x00008199 T __rt_get_argv
...
   # This is also a comment, blank lines are ignored
...
0x0000A4FC D __stdin
0x0000A540 D __stdout
0x0000A584 D __stderr
0xFFFFFFFD N __SIG_IGN
```

Related information

Reading a symdefs file on page 636
Creating a symdefs file on page 635

3.5.6 Edit the symbol tables with a steering file

A steering file is a text file that contains a set of commands to edit the symbol tables of output objects and the dynamic sections of images.

3.5.6.1 Specifying steering files on the linker command-line

You can specify one or more steering files on the linker command-line.

Procedure

Use the option `--edit <file-list>` to specify one or more steering files on the linker command-line.
When you specify more than one steering file, you can use either of the following command-line formats:

```
armlink --edit file1 --edit file2 --edit file3

armlink --edit file1,file2,file3
```

Do not include spaces between the comma and the filenames when using a comma-separated list.

Related information

Steering file command summary on page 639
Steering file format on page 640

3.5.6.2 Steering file command summary

Steering file commands enable you to manage symbols in the symbol table, control the copying of symbols from the static symbol table to the dynamic symbol table, and store information about the libraries that a link unit depends on.

For example, you can use steering files to protect intellectual property, or avoid namespace clashes.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The steering file commands are:

Table 3-15: Steering file command summary

Command	Description
`EXPORT`	Specifies that a symbol can be accessed by other shared objects or executables.
`HIDE`	Makes defined global symbols in the symbol table anonymous.
`IMPORT`	Specifies that a symbol is defined in a shared object at runtime.
`RENAME`	Renames defined and undefined global symbol names.
`REQUIRE`	Creates a `DT_NEEDED` tag in the dynamic array. `DT_NEEDED` tags specify dependencies to other shared objects used by the application, for example, a shared library.
`RESOLVE`	Matches specific undefined references to a defined global symbol.
`SHOW`	Makes global symbols visible. This command is useful if you want to make a specific symbol visible that is hidden using a `HIDE` command with a wildcard.



Note

The steering file commands control only global symbols. Local symbols are not affected by any of these commands.

Related information

Specifying steering files on the linker command-line on page 639
Steering file format on page 640
--edit=file_list on page 455
EXPORT steering file command on page 759
HIDE steering file command on page 761
IMPORT steering file command on page 761
RENAME steering file command on page 763
REQUIRE steering file command on page 764
RESOLVE steering file command on page 764
SHOW steering file command on page 766

3.5.6.3 Steering file format

Each command in a steering file must be on a separate line.

A steering file has the following format:

- Lines with a semicolon (`;`) or hash (`#`) character as the first non-whitespace character are interpreted as comments. A comment is treated as a blank line.
- Blank lines are ignored.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Each non-blank, non-comment line is either a command, or part of a command that is split over consecutive non-blank lines.
- Command lines that end with a comma (`,`) as the last non-whitespace character are continued on the next non-blank line.

Each command line consists of a command, followed by one or more comma-separated operand groups. Each operand group comprises either one or two operands, depending on the command. The command is applied to each operand group in the command. The following rules apply:

- Commands are case-insensitive, but are conventionally shown in uppercase.
- Operands are case-sensitive because they must be matched against case-sensitive symbol names. You can use wildcard characters in operands.

Commands are applied to global symbols only. Other symbols, such as local symbols, are not affected.

The following example shows a sample steering file:

```
; Import my_func1 as func1
IMPORT my_func1 AS func1
# Rename a very long function name to a shorter name
RENAME a_very_long_function_name AS,
       short_func_name
```

Related information

Steering file command summary on page 639
Specifying steering files on the linker command-line on page 639
EXPORT steering file command on page 759
HIDE steering file command on page 761
IMPORT steering file command on page 761
RENAME steering file command on page 763
REQUIRE steering file command on page 764
RESOLVE steering file command on page 764
SHOW steering file command on page 766

3.5.6.4 Hide and rename global symbols with a steering file

You can use a steering file to hide and rename global symbol names in output files.

Use the `HIDE` and `RENAME` commands as required.

For example, you can use steering files to protect intellectual property, or avoid namespace clashes.

Example of renaming a symbol:

`RENAME` steering command example

```
RENAME func1 AS my_func1
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example of hiding symbols:

HIDE steering command example

```
; Hides all global symbols with the 'internal' prefix
HIDE internal*
```

Related information

Specifying steering files on the linker command-line on page 639
Edit the symbol tables with a steering file on page 639
Steering file command summary on page 639
Symdefs file format on page 637
HIDE steering file command on page 761
RENAME steering file command on page 763
--edit=file_list on page 455

3.5.7 Use of $Super$$ and $Sub$$ to patch symbol definitions

There are special patterns that you can use for situations where an existing symbol cannot be modified or recompiled.

An existing symbol cannot be modified if, for example, it is located in an external library or in ROM code. In such cases, you can use the `$Super$$` and `$Sub$$` patterns to patch an existing symbol.

To patch the definition of the function `foo()`, then `$Sub$$foo` and the original definition of `foo()` must be a global or weak definition:

`$Super$$foo`

Identifies the original unpatched function `foo()`. Use this pattern to call the original function directly.

`$Sub$$foo`

Identifies the new function that is called instead of the original function `foo()`. Use this pattern to add processing before or after the original function.

The `$Sub$$` and `$Super$$` linker mechanism can operate only on symbol definitions and references that are visible to the tool. For example, the compiler can replace a call to `printf("Hello\n")` with `puts("Hello")` in a C program. Only the reference to the symbol `puts` is visible to the linker, so defining `$Sub$$printf` does not redirect this call.


Note

- The `$Sub$$` and `$Super$$` mechanism only works at static link time, `$Super$$` references cannot be imported or exported into the dynamic symbol table.
- If the compiler inlines a function, for example `foo()`, then it is not possible to patch the inlined function with the substitute function, `$Sub$$foo`.

The `$Sub$$` and `$Super$$` mechanism interacts with C++ as follows:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The `$Sub$$` and `$Super$$` mechanism works with functions and function templates.
- A `$Sub$$` function cannot be defined for a function template specialization without defining it for the corresponding primary template.
- The `$Sub$$` and `$Super$$` mechanism works with member functions of classes and class templates, but requires the overriding function to be declared in the class definition.
- Using the `$Sub$$` and `$Super$$` mechanism with virtual member functions is unsupported.
- The demangler does not recognize names with `$Sub$$` or `$Super$$` prefixes, and does not affect the debug illusion. It is a library that converts C++ decorated symbols such as `_Z3fooIiEvT_c` into human-readable ones such as `void foo<int>(int, char)`. The demangler does not work correctly with, for example, `$Sub$$_Z3fooIiEvT_c`, in which case the demangler retains the symbol unaltered.

Example: Use $Super$$ and $Sub$$ to insert a call to a function

The following example shows how to use `$Super$$` and `$Sub$$` to insert a call to the function `ExtraFunc()` before the call to the legacy function `foo()`.

```
extern void ExtraFunc(void);
extern void $Super$$foo(void);

/* this function is called instead of the original foo() */
void $Sub$$foo(void)
{
  ExtraFunc();    /* does some extra setup work */
  $Super$$foo();  /* calls the original foo() function */
                  /* To avoid calling the original foo() function
                   * omit the $Super$$foo(); function call.
                   */
}
```

Related information

ELF for the Arm Architecture

3.6 Scatter-loading Features

Describes the scatter-loading features and how you use scatter files with the Arm linker, `armlink`, to create complex images.

3.6.1 The scatter-loading mechanism

The scatter-loading mechanism enables you to specify the memory map of an image to the linker using a description in a text file.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.1.1 Overview of scatter-loading

Scatter-loading gives you complete control over the grouping and placement of image components.

You can use scatter-loading to create simple images, but it is generally only used for images that have a complex memory map. That is, where multiple memory regions are scattered in the memory map at load and execution time.

An image memory map is made up of regions and output sections. Every region in the memory map can have a different load and execution address.

To construct the memory map of an image, the linker must have:

- Grouping information that describes how input sections are grouped into output sections and regions.
- Placement information that describes the addresses where regions are to be located in the memory maps.

When the linker creates an image using a scatter file, it creates some region-related symbols. The linker creates these special symbols only if your code references them.

Related information

When to use scatter-loading on page 644
Scatter file to ELF mapping on page 709
The structure of an Arm ELF image on page 570
Region-related symbols on page 624

3.6.1.2 When to use scatter-loading

Scatter-loading is usually required for implementing embedded systems because these use ROM, RAM, and memory-mapped peripherals.

Situations where scatter-loading is either required or very useful:

Complex memory maps

Code and data that must be placed into many distinct areas of memory require detailed instructions on where to place the sections in the memory space.

Different types of memory

Many systems contain a variety of physical memory devices such as flash, ROM, SDRAM, and fast SRAM. A scatter-loading description can match the code and data with the most appropriate type of memory. For example, interrupt code might be placed into fast SRAM to improve interrupt response time but infrequently-used configuration information might be placed into slower flash memory.

Memory-mapped peripherals

The scatter-loading description can place a data section at a precise address in the memory map so that memory mapped peripherals can be accessed.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Functions at a constant location

A function can be placed at the same location in memory even though the surrounding application has been modified and recompiled. This is useful for jump table implementation.

Using symbols to identify the heap and stack

Symbols can be defined for the heap and stack location when the application is linked.

Related information

Overview of scatter-loading on page 643

3.6.1.3 Linker-defined symbols that are not defined when scatter-loading

When scatter-loading an image, some linker-defined symbols are undefined.

The following symbols are undefined when a scatter file is used:

- `Image$$RO$$Base`.
- `Image$$RO$$Limit`.
- `Image$$RW$$Base`.
- `Image$$RW$$Limit`.
- `Image$$XO$$Base`.
- `Image$$XO$$Limit`.
- `Image$$ZI$$Base`.
- `Image$$ZI$$Limit`.

If you use a scatter file but do not use the special region names for stack and heap, or do not reimplement `__user_setup_stackheap()`, an error message is generated.

Related information

Linker-defined symbols on page 623
Placing the stack and heap with a scatter file on page 645

3.6.1.4 Placing the stack and heap with a scatter file

The Arm C library provides multiple implementations of the function `__user_setup_stackheap()`, and can select the correct one for you automatically from information that is given in a scatter file.

About this task

If you reimplement `__user_setup_stackheap()`, your version does not get invoked when stack and heap are defined in a scatter file.

You might have to update your startup code to use the correct initial stack pointer. Some processors, such as the Cortex®-M3 processor, require that you place the initial stack pointer in

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

the vector table. See *Stack and heap configuration* in Application Note 179 - Cortex-M3 Embedded Software Development for more details.

To use a scatter file to place the stack and heap for the SysV linking model (`--sysv`), you must also use the `armlink` option `--bare_metal_sysv`.

You must ensure correct alignment of the stack and heap:

- In AArch32 state, the stack and heap must be 8-byte aligned.
- In AArch64 state, the stack and heap must be 16-byte aligned.

Procedure

1. Define two special execution regions in your scatter file that are named `ARM_LIB_HEAP` and `ARM_LIB_STACK`.
2. Assign the `EMPTY` attribute to both regions.
   Because the stack and heap are in separate regions, the library selects the non-default implementation of `__user_setup_stackheap()` that uses the value of the symbols:
   - `Image$$ARM_LIB_STACK$$ZI$$Base`.
   - `Image$$ARM_LIB_STACK$$ZI$$Limit`.
   - `Image$$ARM_LIB_HEAP$$ZI$$Base`.
   - `Image$$ARM_LIB_HEAP$$ZI$$Limit`.

   You can specify only one `ARM_LIB_STACK` or `ARM_LIB_HEAP` region, and you must allocate a size.

   ```
   LOAD_FLASH ...
   {
       ...
       ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
       { }
       ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
       { }
       ...
   }
   ```
3. Alternatively, define a single execution region that is named `ARM_LIB_STACKHEAP` to use a combined stack and heap region. Assign the `EMPTY` attribute to the region.
   Because the stack and heap are in the same region, `__user_setup_stackheap()` uses the value of the symbols `Image$$ARM_LIB_STACKHEAP$$ZI$$Base` and `Image$$ARM_LIB_STACKHEAP$$ZI$$Limit`.

Related information

Region-related symbols on page 624
__user_setup_stackheap()

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.1.5 Scatter-loading command-line options

The command-line options to the linker give some control over the placement of data and code, but complete control of placement requires more detailed instructions than can be entered on the command line.

Complex memory maps

Placement of code and data in complex memory maps must be specified in a scatter file. You specify the scatter file with the option:

```
--scatter=<scatter_file>
```

This instructs the linker to construct the image memory map as described in `<scatter_file>`.

Simple memory maps

For simple memory maps, you can place code and data with with the following memory map related command-line options:

- `--partial`.
- `--ro_base`.
- `--rw_base`.
- `--ropi`.
- `--rwpi`.
- `--rosplit`.
- `--split`.
- `--reloc`.
- `--xo_base`
- `--zi_base`.

Note

You cannot use `--scatter` with these options.

Related information

The scatter-loading mechanism on page 643
When to use scatter-loading on page 644
Equivalent scatter-loading descriptions for simple images on page 698
--partial on page 515
--reloc on page 522
--ro_base=address on page 525
--ropi on page 526
--rosplit on page 527

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

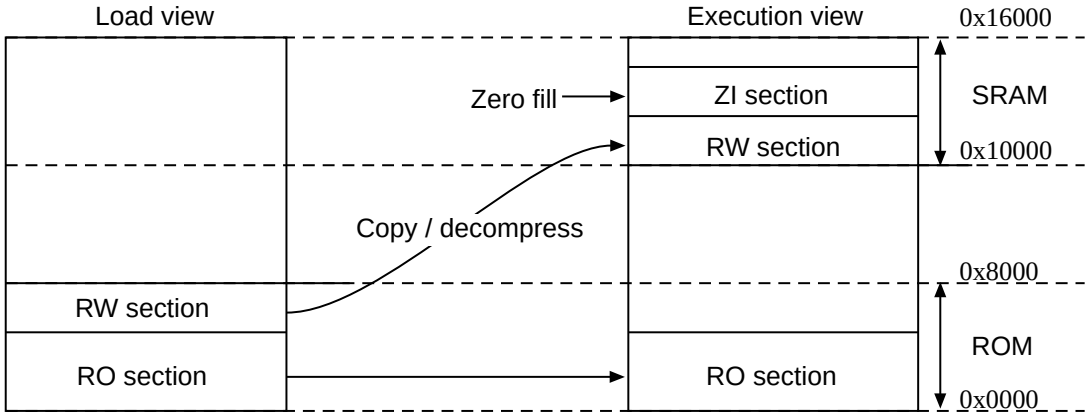
--rw_base=address on page 527
--rwpi on page 528
--scatter=filename on page 529
--split on page 538
--xo_base=address on page 560
--zi_base=address on page 562
Scatter File Syntax on page 711

3.6.1.6 Scatter-loading images with a simple memory map

For images with a simple memory map, you can specify the memory map using only linker command-line options, or with a scatter file.

The following figure shows a simple memory map:

Figure 3-7: Simple scatter-loaded memory map



The following example shows the corresponding scatter-loading description that loads the segments from the object file into memory:

```
LOAD_ROM 0x0000 0x8000        ; Name of load region (LOAD_ROM),
                              ; Start address for load region (0x0000),
                              ; Maximum size of load region (0x8000)
{
    EXEC_ROM 0x0000 0x8000    ; Name of first exec region (EXEC_ROM),
                              ; Start address for exec region (0x0000),
                              ; Maximum size of first exec region (0x8000)
    {
        * (+RO)               ; Place all code and RO data into
                              ; this exec region
    }
    SRAM 0x10000 0x6000       ; Name of second exec region (SRAM),
                              ; Start address of second exec region (0x10000),
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
                                ; Maximum size of second exec region (0x6000)
    {
        * (+RW, +ZI)            ; Place all RW and ZI data into
                                ; this exec region
    }
}
```

The maximum size specifications for the regions are optional. However, if you include them, they enable the linker to check that a region does not overflow its boundary.

Apart from the limit checking, you can achieve the same result with the following linker command-line:

```
armlink --ro_base 0x0 --rw_base 0x10000
```

Related information

Scatter file to ELF mapping on page 709
The scatter-loading mechanism on page 643
When to use scatter-loading on page 644
--ro_base=address on page 525
--rw_base=address on page 527
--xo_base=address on page 560

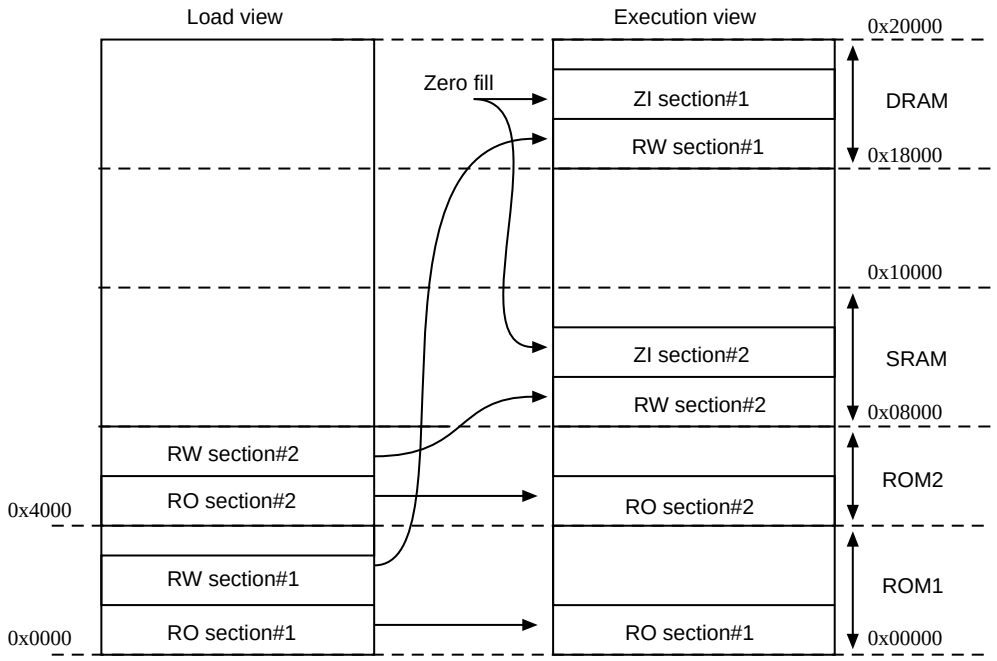
3.6.1.7 Scatter-loading images with a complex memory map

For images with a complex memory map, you cannot specify the memory map using only linker command-line options. Such images require the use of a scatter file.

The following figure shows a complex memory map:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 3-8: Complex memory map



The following example shows the corresponding scatter-loading description that loads the segments from the `program1.o` and `program2.o` files into memory:

```
LOAD_ROM_1 0x0000              ; Start address for first load region (0x0000)
{
    EXEC_ROM_1 0x0000          ; Start address for first exec region (0x0000)
    {
        program1.o (+RO)       ; Place all code and RO data from
                               ; program1.o into this exec region
    }
    DRAM 0x18000 0x8000        ; Start address for this exec region (0x18000),
                               ; Maximum size of this exec region (0x8000)
    {
        program1.o (+RW, +ZI)  ; Place all RW and ZI data from
                               ; program1.o into this exec region
    }
}
LOAD_ROM_2 0x4000              ; Start address for second load region (0x4000)
{
    EXEC_ROM_2 0x4000
    {
        program2.o (+RO)       ; Place all code and RO data from
                               ; program2.o into this exec region
    }
    SRAM 0x8000 0x8000
    {
        program2.o (+RW, +ZI)  ; Place all RW and ZI data from
                               ; program2.o into this exec region
    }
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
}
```



The scatter-loading description in this example specifies the location for code and data for `program1.o` and `program2.o` only. If you link an additional module, for example, `program3.o`, and use this description file, the location of the code and data for program3.o is not specified.

Unless you want to be very rigorous in the placement of code and data, Arm recommends that you use the `*` or `.ANY` specifier to place leftover code and data.

Related information

The scatter-loading mechanism on page 643
Effect of the ABSOLUTE attribute on a root region on page 652
Effect of the FIXED attribute on a root region on page 652
Scatter files containing relative base address load regions and a ZI execution region on page 744
Scatter file to ELF mapping on page 709
When to use scatter-loading on page 644

3.6.2 Root region and the initial entry point

The initial entry point of the image must be in a root region.

If the initial entry point is not in a root region, the link fails and the linker gives an error message.

Example

Root region with the same load and execution address.

```
LR_1 0x040000           ; load region starts at 0x40000
{                       ; start of execution region descriptions
    ER_RO 0x040000      ; load address = execution address
    {
        * (+RO)         ; all RO sections (must include section with
                        ; initial entry point)
    }
    ...                 ; rest of scatter-loading description
}
```

Related information

Effect of the ABSOLUTE attribute on a root region on page 652
Effect of the FIXED attribute on a root region on page 652
Methods of placing functions and data at specific addresses on page 655
Placing functions and data in a named section on page 660
Placement of __at sections at a specific address on page 663
Restrictions on placing __at sections on page 664
Automatic placement of __at sections on page 664
Manual placement of __at sections on page 666

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Place a key in flash memory with an __at section on page 667

3.6.2.1 Effect of the ABSOLUTE attribute on a root region

You can use the ABSOLUTE attribute to specify a root region. This attribute is the default for an execution region.

To specify a root region, use ABSOLUTE as the attribute for the execution region. You can either specify the attribute explicitly or permit it to default, and use the same address for the first execution region and the enclosing load region.

To make the execution region address the same as the load region address, either:

- Specify the same numeric value for both the base address for the execution region and the base address for the load region.
- Specify a +0 offset for the first execution region in the load region.

  If you specify an offset of zero (+0) for all subsequent execution regions in the load region, then all execution regions not following an execution region containing ZI are also root regions.

Example

The following example shows an implicitly defined root region:

```
LR_1 0x040000                   ; load region starts at 0x40000
{                               ; start of execution region descriptions
    ER_RO 0x040000 ABSOLUTE     ; load address = execution address
    {
        * (+RO)                 ; all RO sections (must include the section
                                ; containing the initial entry point)
    }
    ...                         ; rest of scatter-loading description
}
```

Related information

Root region and the initial entry point on page 651
Effect of the FIXED attribute on a root region on page 652
Load region descriptions on page 713
Execution region descriptions on page 720
Considerations when using a relative address +offset for a load region on page 719
Considerations when using a relative address +offset for execution regions on page 729
Load region attributes on page 715
Execution region attributes on page 723
Inheritance rules for load region address attributes on page 717
Inheritance rules for the RELOC address attribute on page 719
Inheritance rules for execution region address attributes on page 727
ENTRY directive on page 991

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.2.2 Effect of the FIXED attribute on a root region

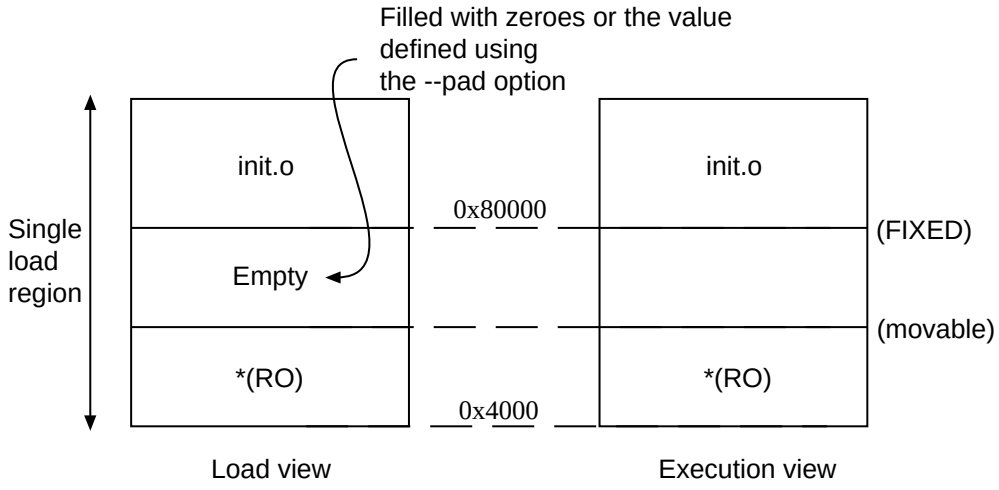
You can use the `FIXED` attribute for an execution region in a scatter file to create root regions that load and execute at fixed addresses.

Use the `FIXED` execution region attribute to ensure that the load address and execution address of a specific region are the same.

You can use the `FIXED` attribute to place any execution region at a specific address in ROM.

For example, the following memory map shows fixed execution regions:

Figure 3-9: Memory map for fixed execution regions



The following example shows the corresponding scatter-loading description:

```
LR_1 0x040000              ; load region starts at 0x40000
{                          ; start of execution region descriptions
    ER_RO 0x040000         ; load address = execution address
    {
        * (+RO)            ; RO sections other than those in init.o
    }
    ER_INIT 0x080000 FIXED ; load address and execution address of this
                           ; execution region are fixed at 0x80000
    {
        init.o(+RO)        ; all RO sections from init.o
    }
    ...                    ; rest of scatter-loading description
}
```

You can use this attribute to place a function or a block of data, for example a constant table or a checksum, at a fixed address in ROM. This makes it easier to access the function or block of data through pointers.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If you place two separate blocks of code or data at the start and end of ROM, some of the memory contents might be unused. For example, you might place some initialization code at the start of ROM and a checksum at the end of ROM. Use the `*` or `.ANY` module selector to flood fill the region between the end of the initialization block and the start of the data block.

To make your code easier to maintain and debug, use the minimum number of placement specifications in scatter files. Leave the detailed placement of functions and data to the linker.


Note

There are some situations where using `FIXED` and a single load region are not appropriate. Other techniques for specifying fixed locations are:

- If your loader can handle multiple load regions, place the RO code or data in its own load region.
- If you do not require the function or data to be at a fixed location in ROM, use `ABSOLUTE` instead of `FIXED`. The loader then copies the data from the load region to the specified address in RAM. `ABSOLUTE` is the default attribute.
- To place a data structure at the location of memory-mapped I/O, use two load regions and specify `UNINIT`. `UNINIT` ensures that the memory locations are not initialized to zero.

Example showing the misuse of the FIXED attribute

The following example shows common cases where the `FIXED` execution region attribute is misused:

```
LR1 0x8000
{
    ER_LOW +0 0x1000
    {
        *(+RO)
    }
; At this point the next available Load and Execution address is 0x8000 + size of
; contents of ER_LOW. The maximum size is limited to 0x1000 so the next available
 Load
; and Execution address is at most 0x9000
    ER_HIGH 0xF0000000 FIXED
    {
        *(+RW,+ZI)
    }
; The required execution address and load address is 0xF0000000. The linker inserts
; 0xF0000000 - (0x8000 + size of(ER_LOW)) bytes of padding so that load address
 matches
; execution address
}
; The other common misuse of FIXED is to give a lower execution address than the
 next
; available load address.
LR_HIGH 0x100000000
{
    ER_LOW 0x1000 FIXED
    {
        *(+RO)
    }
; The next available load address in LR_HIGH is 0x10000000. The required Execution
; address is 0x1000. Because the next available load address in LR_HIGH must
 increase
; monotonically the linker cannot give ER_LOW a Load Address lower than 0x10000000
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
}
```

Related information

Execution region descriptions on page 720
Load region attributes on page 715
Execution region attributes on page 723
Inheritance rules for load region address attributes on page 717
Inheritance rules for the RELOC address attribute on page 719
Inheritance rules for execution region address attributes on page 727

3.6.2.3 Methods of placing functions and data at specific addresses

There are various methods available to place functions and data at specific addresses.

3.6.2.3.1 Placement of functions and data at specific addresses

You can place a single function or data item at a fixed address. You must enable the linker to process the function or data separately from the other input files.

Where they are required, the compiler normally produces RO, RW, and ZI sections from a single source file. These sections contain all the code and data from the source file.



Note

For images targeted at Arm®v6-M, Armv7-M, or Armv8-M, the compiler might generate *eXecute-Only* (XO) sections.

Typically, you create a scatter file that defines an execution region at the required address with a section description that selects only one section.

To place a function or variable at a specific address, it must be placed in its own section. There are several ways to place a function or variable in its own section:

- By default, the compiler places each function and variable in individual ELF sections. To override this default placement, use the `-fno-function-sections` or `-fno-data-sections` compiler options.
- Place the function or data item in its own source file.
- Use `__attribute__((section("<name>")))` to place functions and variables in a specially named section, `.ARM.__at_<address>`, where `<address>` is the address to place the function or variable. For example, `__attribute__((section(".ARM.__at_0x4000")))`.

  To place ZI data at a specific address, use the variable attribute `__attribute__((section("<name>")))` with the special name `.bss.ARM.__at_<address>`

  These specially named sections are called `__at` sections.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Use the `.section` directive from assembly language. In assembly code, the smallest locatable unit is a `.section`.

Related information

Placement of __at sections at a specific address on page 663
Example of how to explicitly place a named section with scatter-loading on page 668
Restrictions on placing __at sections on page 664
--autoat, --no_autoat on page 429
--map, --no_map on page 504
--scatter=filename on page 529
-o filename, --output=filename (armlink) on page 509
AREA directive on page 974

3.6.2.3.2 Placing a variable at a specific address without scatter-loading

This example shows how to modify your source code to place code and data at specific addresses, and does not require a scatter file.

To place code and data at specific addresses without a scatter file:

1. Create the source file `main.c` containing the following code:

```
#include <stdio.h>

extern int sqr(int n1);
const int gValue __attribute__((section(".ARM.__at_0x5000"))) = 3; // Place at
 0x5000
int main(void)
{
    int squared;
    squared=sqr(gValue);
    printf("Value squared is: %d\n", squared);
    return 0;
}
```

2. Create the source file `function.c` containing the following code:

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c function.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --map function.o main.o -o squared.axf
```

   The `--map` option displays the memory map of the image. Also, `--autoat` is the default.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

In this example, `__attribute__((section(".ARM.__AT_0x5000 ")))` specifies that the global variable `gValue` is to be placed at the absolute address `0x5000`. `gValue` is placed in the execution region `ER$$.ARM.__AT_0x5000` and load region `LR$$.ARM.__AT_0x5000`.

The memory map shows:

```
...
  Load Region LR$$.ARM.__AT_0x5000 (Base: 0x00005000, Size: 0x00000004, Max:
0x00000004, ABSOLUTE)

    Execution Region ER$$.ARM.__AT_0x5000 (Base: 0x00005000, Size: 0x00000004, Max:
0x00000004, ABSOLUTE, UNINIT)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00005000   0x00000004   Data   RO           18    .ARM.__AT_0x5000  main.o
```

Related information

--autoat, --no_autoat on page 429
--map, --no_map on page 504
-o filename, --output=filename (armlink) on page 509

3.6.2.3.3 Example of how to place a variable in a named section with scatter-loading

This example shows how to modify your source code to place code and data in a specific section using a scatter file.

To modify your source code to place code and data in a specific section using a scatter file:

1. Create the source file `main.c` containing the following code:

```
#include <stdio.h>

extern int sqr(int n1);
int gSquared __attribute__((section("foo")));  // Place in section foo
int main(void)
{
    gSquared=sqr(3);
    printf("Value squared is: %d\n", gSquared);
    return 0;
}
```

2. Create the source file `function.c` containing the following code:

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. Create the scatter file `scatter.scat` containing the following load region:

```
LR1 0x0000 0x20000
{
    ER1 0x0 0x2000
    {
        *(+RO)                  ; rest of code and read-only data
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    }
    ER2 0x8000 0x2000
    {
        main.o
    }
    ER3 0x10000 0x2000
    {
        function.o
        *(foo)                ; Place gSquared in ER3
    }
    ; RW and ZI data to be placed at 0x200000
    RAM 0x200000 (0x1FF00-0x2000)
    {
        *(+RW, +ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
```

The `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions are required because the program is being linked with the semihosting libraries.

4. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c function.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --map --scatter=scatter.scat function.o main.o -o squared.axf
```

The `--map` option displays the memory map of the image. Also, `--autoat` is the default.

In this example, `__attribute__((section("foo")))` specifies that the global variable `gSquared` is to be placed in a section called `foo`. The scatter file specifies that the section `foo` is to be placed in the `ER3` execution region.

The memory map shows:

```
  Load Region LR1 (Base: 0x00000000, Size: 0x00001570, Max: 0x00020000, ABSOLUTE)

...
    Execution Region ER3 (Base: 0x00010000, Size: 0x00000010, Max: 0x00002000,
 ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00010000   0x0000000c   Code   RO            3    .text
 function.o
    0x0001000c   0x00000004   Data   RW           15    foo                   main.o
...
```


Note

If you omit `*(foo)` from the scatter file, the section is placed in the region of the same type. That is `RAM` in this example.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--autoat, --no_autoat on page 429
--map, --no_map on page 504
-o filename, --output=filename (armlink) on page 509
--scatter=filename on page 529

3.6.2.3.4 Placing a variable at a specific address with scatter-loading

This example shows how to modify your source code to place code and data at a specific address using a scatter file.

To modify your source code to place code and data at a specific address using a scatter file:

1. Create the source file `main.c` containing the following code:

```
#include <stdio.h>

extern int sqr(int n1);
// Place at address 0x10000
const int gValue __attribute__((section(".ARM.__at_0x10000"))) = 3;
int main(void)
{
    int squared;
    squared=sqr(gValue);
    printf("Value squared is: %d\n", squared);
    return 0;
}
```

2. Create the source file `function.c` containing the following code:

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. Create the scatter file `scatter.scat` containing the following load region:

```
LR1 0x0
{
    ER1 0x0
    {
        *(+RO)                      ; rest of code and read-only data
    }
    ER2 +0
    {
        function.o
        *(.ARM.__at_0x10000)        ; Place gValue at 0x10000
    }
    ; RW and ZI data to be placed at 0x200000
    RAM 0x200000 (0x1FF00-0x2000)
    {
        *(+RW, +ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
}
```

The `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions are required because the program is being linked with the semihosting libraries.

4. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c function.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --no_autoat --scatter=scatter.scat --map function.o main.o -o squared.axf
```

The `--map` option displays the memory map of the image.

The memory map shows that the variable is placed in the `ER2` execution region at address `0x10000`:

```
...
    Execution Region ER2 (Base: 0x00002a54, Size: 0x0000d5b0, Max: 0xffffffff,
 ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00002a54   0x0000001c   Code   RO            4    .text.sqr
 function.o
    0x00002a70   0x0000d590   PAD
    0x00010000   0x00000004   Data   RO            9    .ARM.__at_0x10000   main.o
```

In this example, the size of `ER1` is unknown. Therefore, `gValue` might be placed in `ER1` or `ER2`. To make sure that `gValue` is placed in `ER2`, you must include the corresponding selector in `ER2` and link with the `--no_autoat` command-line option. If you omit `--no_autoat`, `gValue` is placed in a separate load region `LR$$.ARM.__at_0x10000` that contains the execution region `ER$$.ARM.__at_0x10000`.

Related information

--autoat, --no_autoat on page 429
--map, --no_map on page 504
-o filename, --output=filename (armlink) on page 509
--scatter=filename on page 529

3.6.2.4 Placing functions and data in a named section

You can place functions and data by separating them into their own objects without having to use toolchain-specific pragmas or attributes. Alternatively, you can specify a name of a section using the function or variable attribute, `__attribute__((section("<name>")))`.

About this task

You can use `__attribute__((section("<name>")))` to place a function or variable in a separate ELF section, where `<name>` is a name of your choice. You can then use a scatter file to place the named sections at specific locations.

You can place ZI data in a named section with `__attribute__((section(".bss.<name>")))`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Use the following procedure to modify your source code to place functions and data in a specific section using a scatter file.

Procedure

1. Create a C source file `file.c` to specify a section name `foo` for a variable and a section name `.bss.mybss` for a zero-initialized variable `z`, for example:

```
#include "stdio.h"

int variable __attribute__((section("foo"))) = 10;
__attribute__((section(".bss.mybss"))) int z;

int main(void)
{
    int x = 4;
    int y = 7;
    z = x + y;
    printf("%d\n",variable);
    printf("%d\n",z);
    return 0;
}
```

2. Create a scatter file to place the named section, `scatter.scat`, for example:

```
LR_1 0x0
{
    ER_RO 0x0 0x4000
    {
        *(+RO)
    }
    ER_RW 0x4000 0x2000
    {
        *(+RW)
    }
    ER_ZI 0x6000 0x2000
    {
        *(+ZI)
    }
    ER_MYBSS 0x8000 0x2000
    {
        *(.bss.mybss)
    }

    ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
    { }
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
}

FLASH 0x24000000 0x4000000
{
    ; rest of code

    ADDER 0x08000000
    {
        file.o (foo)                  ; select section foo from file.o
    }

}
```

The `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions are required because the program is being linked with the semihosting libraries.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

If you omit `file.o (foo)` from the scatter file, the linker places the section in the region of the same type. That is, `ER_RW` in this example.

3. Compile and link the C source:

```
armclang --target=arm-arm-none-eabi -march=armv8-a file.c -g -c -O1 -o file.o
armlink --cpu=8-A.32 --scatter=scatter.scat --map file.o --output=file.axf
```

The `--map` option displays the memory map of the image.

In this example:

- `__attribute__((section("foo")))` specifies that the linker is to place the global variable `variable` in a section called `foo`.
- `__attribute__((section(".bss.mybss")))` specifies that the linker is to place the global variable `z` in a section called `.bss.mybss`.
- The scatter file specifies that the linker is to place the section `foo` in the `ADDER` execution region of the `FLASH` execution region.

The following example shows the output from `--map`:

```
...
    Execution Region ER_MYBSS (Base: 0x00008000, Size: 0x00000004, Max:
 0x00002000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name
 Object

    0x00008000   0x00000004   Zero   RW            7    .bss.mybss
 file.o
...
  Load Region FLASH (Base: 0x24000000, Size: 0x00000004, Max: 0x04000000,
 ABSOLUTE)

    Execution Region ADDER (Base: 0x08000000, Size: 0x00000004, Max: 0xffffffff,
 ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name
 Object

    0x08000000   0x00000004   Data   RW            5    foo
 file.o
...
```


Note

- If scatter-loading is not used, the linker places the section `foo` in the default `ER_RW` execution region of the `LR_1` load region. It also places the section `.bss.mybss` in the default execution region `ER_ZI`.
- If you have a scatter file that does not include the `foo` selector, then the linker places the section in the defined RW execution region.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

You can also place a function at a specific address using `.ARM.__at_<address>` as the section name. For example, to place the function `sqr` at `0x20000`, specify:

```
int sqr(int n1) __attribute__((section(".ARM.__at_0x20000")));

int sqr(int n1)
{
    return n1*n1;
}
```

For more information, see Placement of functions and data at specific addresses.

Related information

Placement of __at sections at a specific address on page 663
Restrictions on placing __at sections on page 664
--autoat, --no_autoat on page 429
--scatter=filename on page 529

3.6.2.5 Placement of __at sections at a specific address

You can give a section a special name that encodes the address where it must be placed.

To place a section at a specific address, use the function or variable attribute `__attribute__((section("<name>")))` with the special name `.ARM.__at_<address>`.

To place ZI data at a specific address, use the variable attribute `__attribute__((section("<name>")))` with the special name `.bss.ARM.__at_<address>`.

`<address>` is the required address of the section. The compiler normalizes this address to eight hexadecimal digits. You can specify the address in hexadecimal or decimal. Sections in the form of `.ARM.__at_<address>` are referred to by the abbreviation `__at`.

The following example shows how to assign a variable to a specific address in C or C++ code:

```
// place variable1 in a section called .ARM.__at_0x8000
int variable1 __attribute__((section(".ARM.__at_0x8000"))) = 10;
```

Note

The name of the section is only significant if you are trying to match the section by name in a scatter file. Without overlays, the linker automatically assigns `__at` sections when you use the `--autoat` command-line option. This option is the default. If you are using overlays, then you cannot use `--autoat` to place `__at` sections.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Supporting arithmetic expressions for an address when placing __at sections

If you need to use an arithmetic expression to specify the section address, then you cannot use the `__attribute__((section(".ARM.__at_<address>")))` attribute. Instead, you must use a pointer approach.

For example, to specify the address as `0xE0001000 + MY_PREDEFINED_OFFSET`, then use the following code:

```
static my_variable_type * const my_address = (my_variable_type *) (0xE0001000 +
 MY_PREDEFINED_OFFSET);

#define my_variable (*my_address)
```

Related information

Placement of functions and data at specific addresses on page 655
Restrictions on placing __at sections on page 664

3.6.2.6 Restrictions on placing __at sections

There are restrictions when placing `__at` sections at specific addresses.

The following restrictions apply:

- `__at` section address ranges must not overlap, unless the overlapping sections are placed in different overlay regions.
- `__at` sections are not permitted in position independent execution regions.
- You must not reference the linker-defined symbols `$$Base`, `$$Limit` and `$$Length` of an `__at` section.
- `__at` sections must not be used in *Base Platform Application Binary Interface* (BPABI) executables and BPABI *dynamically linked libraries* (DLLs).
- `__at` sections must have an address that is a multiple of their alignment.
- `__at` sections ignore any `+FIRST` or `+LAST` ordering constraints.

Related information

Placement of __at sections at a specific address on page 663

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.2.7 Automatic placement of __at sections

The automatic placement of `__at` sections is enabled by default. Use the linker command-line option, `--no_autoat` to disable this feature.


Note

You cannot use `__at` section placement with position independent execution regions.

When linking with the `--autoat` option, the linker does not place `__at` sections with scatter-loading selectors. Instead, the linker places the `__at` section in a compatible region. If no compatible region is found, the linker creates a load region and an execution region for the `__at` section.

All linker execution regions created by `--autoat` have the `UNINIT` scatter-loading attribute. If you require a Zero-Initialized (ZI) `__at` section to be zero-initialized, then it must be placed within a compatible region. A linker execution region created by `--autoat` must have a base address that is at least 4 byte-aligned. If any region is incorrectly aligned, the linker reports an error.

A compatible region is one where:

- The `__at` address lies within the execution region base and limit, where limit is the base address + maximum size of execution region. If no maximum size is set, the linker sets the limit for placing `__at` sections as the current size of the execution region without `__at` sections plus a constant. The default value of this constant is `10240` bytes, but you can change the value using the `--max_er_extension` command-line option.
- The execution region meets at least one of the following conditions:
  - It has a selector that matches the `__at` section by the standard scatter-loading rules.
  - It has at least one section of the same type (RO or RW) as the `__at` section.
  - It does not have the `EMPTY` attribute.


Note

The linker considers an `__at` section with type RW compatible with RO.

The following example shows the sections `.ARM.__at_0x0000` type RO, `.ARM.__at_0x4000` type RW, and `.ARM.__at_0x8000` type RW:

```
// place the RO variable in a section called .ARM.__at_0x0000
const int foo __attribute__((section(".ARM.__at_0x0000"))) = 10;

// place the RW variable in a section called .ARM.__at_0x4000
int bar __attribute__((section(".ARM.__at_0x4000"))) = 100;

// place "variable" in a section called .ARM.__at_0x8000
int variable __attribute__((section(".ARM.__at_0x8000")));
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The following scatter file shows how automatically to place these __at sections:

```
LR1 0x0
{
    ER_RO 0x0 0x4000
    {
        *(+RO)      ; .ARM.__at_0x0000 lies within the bounds of ER_RO
    }
    ER_RW 0x4000 0x2000
    {
        *(+RW)      ; .ARM.__at_0x4000 lies within the bounds of ER_RW
    }
    ER_ZI 0x6000 0x2000
    {
        *(+ZI)
    }
}
; The linker creates a load region and an execution region for the __at section
; .ARM.__at_0x8000 because it lies outside all candidate regions.
```

Related information

Placement of __at sections at a specific address on page 663
Manual placement of __at sections on page 666
Place a key in flash memory with an __at section on page 667
Execution region descriptions on page 720
Placing functions and data in a named section on page 660
Restrictions on placing __at sections on page 664
--autoat, --no_autoat on page 429
--ro_base=address on page 525
--rw_base=address on page 527
--xo_base=address on page 560
--zi_base=address on page 562
Execution region attributes on page 723
--max_er_extension=size on page 505
__attribute__((section("name"))) variable attribute

3.6.2.8 Manual placement of __at sections

You can have direct control over the placement of __at sections, if required.

You can use the standard section-placement rules to place __at sections when using the --no_autoat command-line option.

Note

You cannot use __at section placement with position independent execution regions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The following example shows the placement of read-only sections `.ARM.__at_0x2000` and the read-write section `.ARM.__at_0x4000`. Load and execution regions are not created automatically in manual mode. An error is produced if an `__at` section cannot be placed in an execution region.

The following example shows the placement of the variables in C or C++ code:

```
// place the RO variable in a section called .ARM.__at_0x2000
const int foo __attribute__((section(".ARM.__at_0x2000"))) = 100;
// place the RW variable in a section called .ARM.__at_0x4000
int bar __attribute__((section(".ARM.__at_0x4000")));
```

The following scatter file shows how to place `__at` sections manually:

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)               ; .ARM.__at_0x0000 is selected by +RO
    }
    ER_RO2 0x2000
    {
        *(.ARM.__at_0x02000)  ; .ARM.__at_0x2000 is selected by the section named
                              ; .ARM.__at_0x2000
    }
    ER2 0x4000
    {
        *(+RW, +ZI)           ; .ARM.__at_0x4000 is selected by +RW
    }
}
```

Related information

Placement of __at sections at a specific address on page 663
Automatic placement of __at sections on page 664
Place a key in flash memory with an __at section on page 667
Execution region descriptions on page 720
Placing functions and data in a named section on page 660
Restrictions on placing __at sections on page 664
--autoat, --no_autoat on page 429
Execution region attributes on page 723
__attribute__((section("name"))) variable attribute

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.2.9 Place a key in flash memory with an __at section

Some flash devices require a key to be written to an address to activate certain features. An `__at` section provides a simple method of writing a value to a specific address.

Placing the flash key variable in C or C++ code

Assume that a device has flash memory from `0x8000` to `0x10000` and a key is required in address `0x8000`. To do this with an `__at` section, you must declare a variable so that the compiler can generate a section called `.ARM.__at_0x8000`.

```
// place flash_key in a section called .ARM.__at_0x8000

long flash_key __attribute__((section(".ARM.__at_0x8000")));
```

Manually placing a flash execution region

The following example shows how to manually place a flash execution region with a scatter file:

```
ER_FLASH 0x8000 0x2000
{
    *(+RW)
    *(.ARM.__at_0x8000) ; key
}
```

Use the linker command-line option `--no_autoat` to enable manual placement.

Automatically placing a flash execution region

The following example shows how to automatically place a flash execution region with a scatter file. Use the linker command-line option `--autoat` to enable automatic placement.

```
LR1 0x0
{
    ER_FLASH 0x8000 0x2000
    {
        *(+RO)                    ; other code and read-only data, the
                                  ; __at section is automatically selected
    }
    ER2 0x4000
    {
        *(+RW +ZI)                ; Any other RW and ZI variables
    }
}
```

Related information

Placement of __at sections at a specific address on page 663
Automatic placement of __at sections on page 664
Manual placement of __at sections on page 666
Execution region descriptions on page 720
--autoat, --no_autoat on page 429
Section placement with the FIRST and LAST attributes on page 594

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.3 Example of how to explicitly place a named section with scatter-loading

This example shows how to place a named section explicitly using scatter-loading.

Consider the following source files, `init.c` and `data.c`:

```
//init.c
//------
int foo() __attribute__((section("INIT")));
int foo() {
  return 1;
}

int bar() {
  return 2;
}

//data.c
//------
const long padding=123;
int z=5;
```

The following scatter file shows how to place a named section explicitly:

```
LR1 0x0 0x10000
{
    ; Root Region, containing init code
    ER1 0x0 0x2000
    {
        init.o (INIT, +FIRST)   ; place init code at exactly 0x0
        *(+RO)                  ; rest of code and read-only data
    }
    ; RW and ZI data to be placed at 0x400000
    RAM_RW 0x400000 (0x1FF00-0x2000)
    {
        *(+RW)
    }
    RAM_ZI +0
    {
        *(+ZI)
    }
    ; execution region at 0x1FF00
    ; maximum space available for table is 0xFF
    DATABLOCK 0x1FF00 0xFF
    {
        data.o(+RO-DATA)  ; place RO data between 0x1FF00 and 0x1FFFE
    }
}
```

In this example, the scatter-loading description places:

- The initialization code is placed in the `INIT` section in the init.o file. This example shows that the code from the `INIT` section is placed first, at address `0x0`, followed by the remainder of the RO code and all of the RO data except for the RO data in the object `data.o`.
- All global RW variables in RAM at `0x400000`.
- A table of `RO-DATA` from `data.o` at address `0x1FF00`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The resulting image memory map is as follows:

```
Memory Map of the image

  Image entry point : Not specified.

  Load Region LR1 (Base: 0x00000000, Size: 0x00000018, Max: 0x00010000, ABSOLUTE)

    Execution Region ER1 (Base: 0x00000000, Size: 0x00000010, Max: 0x00002000,
ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000000   0x00000008   Code   RO            4    INIT                init.o
    0x00000008   0x00000008   Code   RO            1    .text               init.o
    0x00000010   0x00000000   Code   RO           16    .text               data.o


    Execution Region DATABLOCK (Base: 0x0001ff00, Size: 0x00000004, Max: 0x000000ff,
ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x0001ff00   0x00000004   Data   RO           19    .rodata             data.o


    Execution Region RAM_RW (Base: 0x00400000, Size: 0x00000004, Max: 0x0001df00,
ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00400000   0x00000000   Data   RW            2    .data               init.o
    0x00400000   0x00000004   Data   RW           17    .data               data.o


    Execution Region RAM_ZI (Base: 0x00400004, Size: 0x00000000, Max: 0xffffffff,
ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00400004   0x00000000   Zero   RW            3    .bss                init.o
    0x00400004   0x00000000   Zero   RW           18    .bss                data.o
```

Related information

Effect of the FIXED attribute on a root region on page 652
Load region descriptions on page 713
Execution region descriptions on page 720
Load region attributes on page 715
Execution region attributes on page 723
Inheritance rules for load region address attributes on page 717
Inheritance rules for the RELOC address attribute on page 719
Inheritance rules for execution region address attributes on page 727
ENTRY directive on page 991

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.4 Manual placement of unassigned sections

The linker attempts to place Input sections into specific execution regions. For any Input sections that cannot be resolved, and where the placement of those sections is not important, you can specify where the linker is to place them.

To place sections that are not automatically assigned to specific execution regions, use the `.ANY` module selector in a scatter file.

Usually, a single `.ANY` selector is equivalent to using the `*` module selector. However, unlike `*`, you can specify `.ANY` in multiple execution regions.

The linker has default rules for placing unassigned sections when you specify multiple `.ANY` selectors. You can override the default rules using the following command-line options:

- `--any_contingency` to permit extra space in any execution regions containing `.ANY` sections for linker-generated content such as veneers and alignment padding.
- `--any_placement` to provide more control over the placement of unassigned sections.
- `--any_sort_order` to control the sort order of unassigned Input sections.


Note

The placement of data can cause some data to be removed and shrink the size of the sections.

In a scatter file, you can also:

- Assign a priority to a `.ANY` selector to give you more control over how the unassigned sections are divided between multiple execution regions. You can assign the same priority to more than one execution region.
- Specify the maximum size for an execution region that the linker can fill with unassigned sections.

The following are relevant operations in the linking process and their order:

1. `.ANY` placement.
2. String merging.
3. Region table creation.
4. Late library load (scatter-load functions).
5. Veneer generation + literal pool merging.

String and literal pool merging can reduce execution size, while region table creation, late library load, and veneer generation can increase it. Padding also affects the execution size of the region.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

Extra, more-specific operations can also increase or decrease execution size after the `.ANY` placement, such as the generation of PLT/GOT and exception-section optimizations.

3.6.4.1 Default rules for placing unassigned sections

The linker has default rules for placing sections when using multiple `.ANY` selectors.

When more than one `.ANY` selector is present in a scatter file, the linker sorts sections in descending size order. It then takes the unassigned section with the largest size and assigns the section to the most specific `.ANY` execution region that has enough free space. For example, `.ANY(.text)` is judged to be more specific than `.ANY(+RO)`.

If several execution regions are equally specific, then the section is assigned to the execution region with the most available remaining space.

For example:

- You might have two equally specific execution regions where one has a size limit of `0x2000` and the other has no limit. In this case, all the sections are assigned to the second unbounded `.ANY` region.
- You might have two equally specific execution regions where one has a size limit of `0x2000` and the other has a size limit of `0x3000`. In this case, the first sections to be placed are assigned to the second `.ANY` region of size limit `0x3000`. This assignment continues until the remaining size of the second `.ANY` region is reduced to `0x2000`. From this point, sections are assigned alternately between both `.ANY` execution regions.

You can specify a maximum amount of space to use for unassigned sections with the execution region attribute `ANY_SIZE`.

Related information

How the linker resolves multiple matches when processing scatter files on page 706
--any_placement=algorithm on page 425
--any_contingency on page 425

3.6.4.2 Command-line options for controlling the placement of unassigned sections

You can modify how the linker places unassigned input sections when using multiple `.ANY` selectors by using a different placement algorithm or a different sort order.

The following command-line options are available:

- `--any_placement=<algorithm>`, where `<algorithm>` is one of `first_fit`, `worst_fit`, `best_fit`, or `next_fit`.
- `--any_sort_order=<order>`, where `<order>` is one of `cmdline` or `descending_size`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Use `first_fit` when you want to fill regions in order.

Use `best_fit` when you want to fill regions to their maximum.

Use `worst_fit` when you want to fill regions evenly. With equal sized regions and sections `worst_fit` fills regions cyclically.

Use `next_fit` when you need a more deterministic fill pattern.

If the linker attempts to fill a region to its limit, as it does with `first_fit` and `best_fit`, it might overfill the region. This is because linker-generated content such as padding and veneers are not known until sections have been assigned to `.ANY` selectors. If this occurs you might see the following error:

```
Error: L6220E: Execution region <regionname> size (<size> bytes) exceeds limit (<limit> bytes).
```

The `--any_contingency` option prevents the linker from filling the region up to its maximum. It reserves a portion of the region's size for linker-generated content and fills this contingency area only if no other regions have space. It is enabled by default for the `first_fit` and `best_fit` algorithms, because they are most likely to exhibit this behavior.

Related information

Examples of using placement algorithms for .ANY sections on page 675
Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 677
Examples of using sorting algorithms for .ANY sections on page 679
Behavior when .ANY sections overflow because of linker-generated content on page 681
--any_sort_order=order on page 427
--map, --no_map on page 504
armlink Command-line Options on page 425
--tiebreaker=option on page 550
--any_placement=algorithm on page 425
--any_contingency on page 425

3.6.4.3 Prioritizing the placement of unassigned sections

You can give a priority ordering when placing unassigned sections with multiple `.ANY` module selectors.

Procedure

To prioritize the order of multiple `.ANY` sections use the `.ANY<num>` selector, where `<num>` is a positive integer starting at zero.
The highest priority is given to the selector with the highest integer.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The following example shows how to use `.ANY<num>`:

```
lr1 0x8000 1024
{
    er1 +0 512
    {
        .ANY1(+RO) ; evenly distributed with er3
    }
    er2 +0 256
    {
        .ANY2(+RO) ; Highest priority, so filled first
    }
    er3 +0 256
    {
        .ANY1(+RO) ; evenly distributed with er1
    }
}
```

Related information

Examples of using placement algorithms for .ANY sections on page 675
Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 677
Examples of using sorting algorithms for .ANY sections on page 679
Behavior when .ANY sections overflow because of linker-generated content on page 681
--any_sort_order=order on page 427
--map, --no_map on page 504
armlink Command-line Options on page 425
--tiebreaker=option on page 550
How the linker resolves multiple matches when processing scatter files on page 706

3.6.4.4 Specify the maximum region size permitted for placing unassigned sections

You can specify the maximum size in a region that `armlink` can fill with unassigned sections.

Use the execution region attribute `ANY_SIZE <max_size>` to specify the maximum size in a region that `armlink` can fill with unassigned sections.

Be aware of the following restrictions when using this keyword:

- `<max_size>` must be less than or equal to the region size.
- If you use `ANY_SIZE` on a region without a `.ANY` selector, it is ignored by `armlink`.

When `ANY_SIZE` is present, `armlink` does not attempt to calculate contingency and strictly follows the `.ANY` priorities.

When `ANY_SIZE` is not present for an execution region containing a `.ANY` selector, and you specify the `--any_contingency` command-line option, then `armlink` attempts to adjust the contingency for that execution region. The aims are to:

- Never overflow a `.ANY` region.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Make sure there is a contingency reserved space left in the given execution region. This space is reserved for veneers and section padding.

If you specify `--any_contingency` on the command line, it is ignored for regions that have `ANY_SIZE` specified. It is used as normal for regions that do not have `ANY_SIZE` specified.

Example

The following example shows how to use `ANY_SIZE`:

```
LOAD_REGION 0x0 0x3000
{
    ER_1 0x0 ANY_SIZE 0xF00 0x1000
    {
        .ANY
    }
    ER_2 0x0 ANY_SIZE 0xFB0 0x1000
    {
        .ANY
    }
    ER_3 0x0 ANY_SIZE 0x1000 0x1000
    {
        .ANY
    }
}
```

In this example:

- `ER_1` has `0x100` reserved for linker-generated content.
- `ER_2` has `0x50` reserved for linker-generated content. That is about the same as the automatic contingency of `--any_contingency`.
- `ER_3` has no reserved space. Therefore, 100% of the region is filled, with no contingency for veneers. Omitting the `ANY_SIZE` parameter causes 98% of the region to be filled, with a two percent contingency for veneers.

Related information

Examples of using placement algorithms for .ANY sections on page 675
Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 677
Examples of using sorting algorithms for .ANY sections on page 679
Behavior when .ANY sections overflow because of linker-generated content on page 681
--any_sort_order=order on page 427
--map, --no_map on page 504
--any_contingency on page 425

3.6.4.5 Examples of using placement algorithms for .ANY sections

These examples show the operation of the placement algorithms for `RO-CODE` sections in `sections.o`.

The input section properties and ordering are shown in the following table:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 3-16: Input section properties for placement of .ANY sections

Name	Size (bytes)
sec1	0x4
sec2	0x4
sec3	0x4
sec4	0x4
sec5	0x4
sec6	0x4

The scatter file that the examples use is:

```
LR 0x100
{
  ER_1 0x100 0x10
  {
     .ANY
  }
  ER_2 0x200 0x10
  {
     .ANY
  }
}
```


Note

These examples have `--any_contingency` disabled.

Example for first_fit, next_fit, and best_fit

This example shows the image memory map where several sections of equal size are assigned to two regions with one selector. The selectors are equally specific, equivalent to `.ANY(+RO)` and have no priority.

```
Execution Region ER_1 (Base: 0x00000100, Size: 0x00000010, Max: 0x00000010,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000100   0x00000004   Code   RO            1    sec1               sections.o
0x00000104   0x00000004   Code   RO            2    sec2               sections.o
0x00000108   0x00000004   Code   RO            3    sec3               sections.o
0x0000010c   0x00000004   Code   RO            4    sec4               sections.o


Execution Region ER_2 (Base: 0x00000200, Size: 0x00000008, Max: 0x00000010,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000200   0x00000004   Code   RO            5    sec5               sections.o
0x00000204   0x00000004   Code   RO            6    sec6               sections.o
```

In this example:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- For `first_fit`, the linker first assigns all the sections it can to `ER_1`, then moves on to `ER_2` because that is the next available region.
- For `next_fit`, the linker does the same as `first_fit`. However, when `ER_1` is full it is marked as `FULL` and is not considered again. In this example, `ER_1` is full. `ER_2` is then considered.
- For `best_fit`, the linker assigns `sec1` to `ER_1`. It then has two regions of equal priority and specificity, but `ER_1` has less space remaining. Therefore, the linker assigns `sec2` to `ER_1`, and continues assigning sections until `ER_1` is full.

Example for worst_fit

This example shows the image memory map when using the `worst_fit` algorithm.

```
Execution Region ER_1 (Base: 0x00000100, Size: 0x0000000c, Max: 0x00000010,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000100   0x00000004   Code   RO            1    sec1                sections.o
0x00000104   0x00000004   Code   RO            3    sec3                sections.o
0x00000108   0x00000004   Code   RO            5    sec5                sections.o


Execution Region ER_2 (Base: 0x00000200, Size: 0x0000000c, Max: 0x00000010,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000200   0x00000004   Code   RO            2    sec2                sections.o
0x00000204   0x00000004   Code   RO            4    sec4                sections.o
0x00000208   0x00000004   Code   RO            6    sec6                sections.o
```

The linker first assigns `sec1` to `ER_1`. It then has two equally specific and priority regions. It assigns `sec2` to the one with the most free space, `ER_2` in this example. The regions now have the same amount of space remaining, so the linker assigns `sec3` to the first one that appears in the scatter file, that is `ER_1`.



Note

The behavior of `worst_fit` is the default behavior in this version of the linker, and it is the only algorithm available in earlier linker versions.

Related information

Prioritizing the placement of unassigned sections on page 673
Command-line options for controlling the placement of unassigned sections on page 672
Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 677
--scatter=filename on page 529
Specify the maximum region size permitted for placing unassigned sections on page 674

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority

This example shows the operation of the `next_fit` placement algorithm for `RO-CODE` sections in `sections.o`.

The input section properties and ordering are shown in the following table:

Table 3-17: Input section properties for placement of sections with next_fit

Name	Size
sec1	0x14
sec2	0x14
sec3	0x10
sec4	0x4
sec5	0x4
sec6	0x4

The scatter file used for the examples is:

```
LR 0x100
{
  ER_1 0x100 0x20
  {
     .ANY1(+RO-CODE)
  }
  ER_2 0x200 0x20
  {
     .ANY2(+RO)
  }
  ER_3 0x300 0x20
  {
     .ANY3(+RO)
  }
}
```



Note

This example has `--any_contingency` disabled.

The `next_fit` algorithm is different to the others in that it never revisits a region that is considered to be full. This example also shows the interaction between priority and specificity of selectors. This is the same for all the algorithms.

```
Execution Region ER_1 (Base: 0x00000100, Size: 0x00000014, Max: 0x00000020,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000100   0x00000014   Code   RO            1    sec1               sections.o
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
Execution Region ER_2 (Base: 0x00000200, Size: 0x0000001c, Max: 0x00000020,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000200   0x00000010   Code   RO            3    sec3                sections.o
0x00000210   0x00000004   Code   RO            4    sec4                sections.o
0x00000214   0x00000004   Code   RO            5    sec5                sections.o
0x00000218   0x00000004   Code   RO            6    sec6                sections.o


Execution Region ER_3 (Base: 0x00000300, Size: 0x00000014, Max: 0x00000020,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000300   0x00000014   Code   RO            2    sec2                sections.o
```

In this example:

- The linker places `sec1` in `ER_1` because `ER_1` has the most specific selector. `ER_1` now has `0x6` bytes remaining.
- The linker then tries to place `sec2` in `ER_1`, because it has the most specific selector, but there is not enough space. Therefore, `ER_1` is marked as full and is not considered in subsequent placement steps. The linker chooses `ER_3` for `sec2` because it has higher priority than `ER_2`.
- The linker then tries to place `sec3` in `ER_3`. It does not fit, so `ER_3` is marked as full and the linker places `sec3` in `ER_2`.
- The linker now processes `sec4`. This is `0x4` bytes so it can fit in either `ER_1` or `ER_3`. Because both of these sections have previously been marked as full, they are not considered. The linker places all remaining sections in `ER_2`.
- If another section `sec7` of size `0x8` exists, and is processed after `sec6` the example fails to link. The algorithm does not attempt to place the section in `ER_1` or `ER_3` because they have previously been marked as full.

Related information

Specify the maximum region size permitted for placing unassigned sections on page 674
Prioritizing the placement of unassigned sections on page 673
Command-line options for controlling the placement of unassigned sections on page 672
Examples of using placement algorithms for .ANY sections on page 675
How the linker resolves multiple matches when processing scatter files on page 706
Behavior when .ANY sections overflow because of linker-generated content on page 681
--scatter=filename on page 529

3.6.4.7 Examples of using sorting algorithms for .ANY sections

These examples show the operation of the sorting algorithms for `RO-CODE` sections in `sections_a.o` and `sections_b.o`.

The input section properties and ordering are shown in the following table:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

sections_a.o		sections_b.o	
Name	Size	Name	Size
seca_1	0x4	secb_1	0x4
seca_2	0x4	secb_2	0x4
seca_3	0x10	secb_3	0x10
seca_4	0x14	secb_4	0x14

Descending size example

The following linker command-line options are used for this example:

```
--any_sort_order=descending_size sections_a.o sections_b.o --scatter scatter.txt
```

The following table shows the order that the sections are processed by the `.ANY` assignment algorithm.

Table 3-19: Sort order for descending_size algorithm

Name	Size
seca_4	0x14
secb_4	0x14
seca_3	0x10
secb_3	0x10
seca_1	0x4
seca_2	0x4
secb_1	0x4
secb_2	0x4

With `--any_sort_order=descending_size`, sections of the same size use the creation index as a tiebreaker.

Command-line example

The following linker command-line options are used for this example:

```
--any_sort_order=cmdline sections_a.o sections_b.o --scatter scatter.txt
```

The following table shows the order that the sections are processed by the `.ANY` assignment algorithm.

Table 3-20: Sort order for cmdline algorithm

Name	Size
seca_1	0x4
seca_2	0x4
seca_3	0x10
seca_4	0x14

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Name	Size
secb_1	0x4
secb_2	0x4
secb_3	0x10
secb_4	0x14

That is, the input sections are sorted by command-line index.

Related information

Prioritizing the placement of unassigned sections on page 673
Command-line options for controlling the placement of unassigned sections on page 672
--any_sort_order=order on page 427
--scatter=filename on page 529
Specify the maximum region size permitted for placing unassigned sections on page 674

3.6.4.8 Behavior when .ANY sections overflow because of linker-generated content

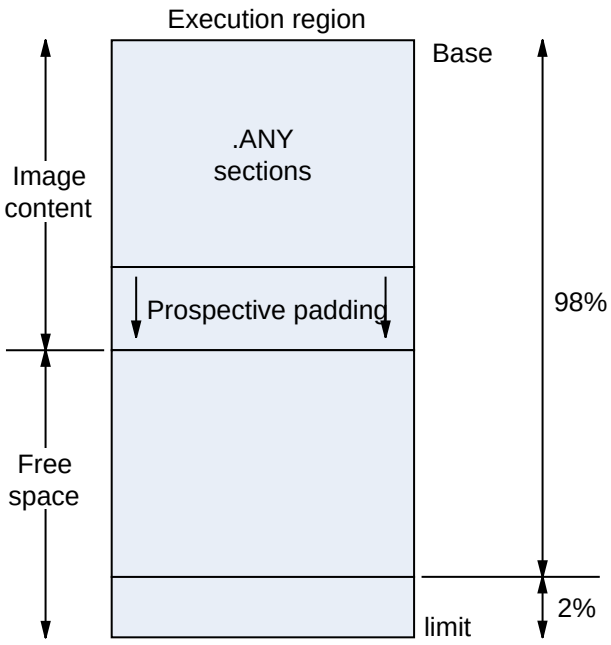
Because linker-generated content might cause `.ANY` sections to overflow, a contingency algorithm is included in the linker.

The linker does not know the address of a section until it is assigned to a region. Therefore, when filling `.ANY` regions, the linker cannot calculate the contingency space and cannot determine if calling functions require veneers. The linker provides a contingency algorithm that gives a worst-case estimate for padding and an extra two percent for veneers. To enable this algorithm, use the `--any_contingency` command-line option.

The following diagram represents an example image layout during `.ANY` placement:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 3-10: .ANY contingency



The downward arrows for prospective padding show that the prospective padding continues to grow as more sections are added to the `.ANY` selector.

Prospective padding is dealt with before the two percent veneer contingency.

When the prospective padding is cleared, the priority is set to zero. When the two percent is cleared, the priority is decremented again.

You can also use the `ANY_SIZE` keyword on an execution region to specify the maximum amount of space in the region to set aside for `.ANY` section assignments.

You can use the `armlink` command-line option `--info=any` to get extra information on where the linker has placed sections. This information can be useful when trying to debug problems.

Note

When there is only one `.ANY` selector, it might not behave identically to `*`. The algorithms that are used to determine the size of the section and place data still run with `.ANY` and they try to estimate the impact of changes that might affect the size of sections. These algorithms do not run if `*` is used instead. When it is appropriate to use one or the other of `.ANY` or `*`, then you must not use a single `.ANY` selector that applies to a kind of data, such as RO, RW, or ZI. For example, `.ANY (+RO)`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

You might see error `L6407E` generated, for example:

```
Error: L6407E: Sections of aggregate size 0x128 bytes could not fit
 into .ANY selector(s).
```

However, increasing the section size by `0x128` bytes does not necessarily lead to a successful link. The failure to link is because of the extra data, such as region table entries, that might end up in the region after adding more sections.

Example

1. Create the following `foo.c` program:

```
#include "stdio.h"

int array[10] __attribute__ ((section ("ARRAY")));

struct S {
    char A[8];
    char B[4];
};
struct S s;

struct S* get()
{
    return &s;
}

int sqr(int n1);

int gSquared __attribute__((section(".ARM.__at_0x5000")));  // Place at 0x5000

int sqr(int n1)
{
    return n1*n1;
}

int main(void) {
    int i;
    for (i=0; i<10; i++) {
        array[i]=i*i;
        printf("%d\n", array[i]);
    }
    gSquared=sqr(i);
    printf("%d squared is: %d\n", i, gSquared);

    return sizeof(array);
}
```

2. Create the following `scatter.scat` file:

```
LOAD_REGION 0x0 0x3000
{
   ER_1 0x0 0x1000
   {
      .ANY
   }
   ER_2 (ImageLimit(ER_1)) 0x1500
   {
      .ANY
   }
   ER_3 (ImageLimit(ER_2)) 0x500
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    {
       .ANY
    }
    ER_4 (ImageLimit(ER_3)) 0x1000
    {
        *(+RW,+ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
```

3. Compile and link the program as follows:

```
armclang -c --target=arm-arm-none-eabi -mcpu=cortex-m4 -o foo.o foo.c
armlink --cpu=cortex-m4 --any_contingency --scatter=scatter.scat --info=any -o
 foo.axf foo.o
```

The following shows an example of the information generated:

```
================================================================================


Sorting unassigned sections by descending size for .ANY placement.
Using Worst Fit .ANY placement algorithm.
.ANY contingency enabled.

Exec Region    Event                          Idx          Size        Section Name
                Object
ER_2           Assignment: Worst fit          144          0x0000041a  .text
                c_wu.l(_printf_fp_dec.o)
ER_2           Assignment: Worst fit          261          0x00000338  CL$
$btod_div_common           c_wu.l(btod.o)
ER_1           Assignment: Worst fit          146          0x000002fc  .text
                c_wu.l(_printf_fp_hex.o)
ER_2           Assignment: Worst fit          260          0x00000244  CL$
$btod_mult_common          c_wu.l(btod.o)
...
ER_1           Assignment: Worst fit          3            0x00000090  .text
                foo.o
...
ER_3           Assignment: Worst fit          100          0x0000000a
.ARM.Collect$$_printf_percent$$00000007    c_wu.l(_printf_ll.o)
ER_3           Info: .ANY limit reached       -            -           -
                -
ER_1           Assignment: Highest priority   423          0x0000000a  .text
                c_wu.l(defsig_exit.o)
...
.ANY contingency summary
Exec Region    Contingency     Type
ER_1           161             Auto
ER_2           180             Auto
ER_3           73              Auto

================================================================================


Sorting unassigned sections by descending size for .ANY placement.
Using Worst Fit .ANY placement algorithm.
.ANY contingency enabled.

Exec Region    Event                          Idx          Size        Section Name
                Object
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
ER_2            Info: .ANY limit reached      -           -           -
                  -
ER_1            Info: .ANY limit reached      -           -           -
                  -
ER_3            Info: .ANY limit reached      -           -           -
                  -
ER_2            Assignment: Worst fit         533         0x00000034  !!!scatter
                  c_wu.l(__scatter.o)
ER_2            Assignment: Worst fit         535         0x0000001c  !!handler_zi
                  c_wu.l(__scatter_zi.o)
```

Related information

--any_contingency on page 425
Prioritizing the placement of unassigned sections on page 673
Command-line options for controlling the placement of unassigned sections on page 672
Specify the maximum region size permitted for placing unassigned sections on page 674
--info=topic[,topic,...] (armlink) on page 474
Syntax of an input section description on page 730
Execution region attributes on page 723

3.6.5 Placing veneers with a scatter file

You can place veneers at a specific location with a linker-generated symbol. Veneers allow switching between A32 and T32 code or allow a longer program jump than can be specified in a single instruction.

Procedure

To place veneers at a specific location, include the linker-generated symbol `Veneer$$Code` in a scatter file. At most, one execution region in the scatter file can have the `*(Veneer$$Code)` section selector.
If it is safe to do so, the linker places veneer input sections into the region identified by the `*(Veneer$$Code)` section selector. It might not be possible for a veneer input section to be assigned to the region because of address range problems or execution region size limitations. If the veneer cannot be added to the specified region, it is added to the execution region containing the relocated input section that generated the veneer.

Note

Instances of `*(IWV$$Code)` in scatter files from earlier versions of Arm® tools are automatically translated into `*(Veneer$$Code)`. Use `*(Veneer$$Code)` in new descriptions.

`*(Veneer$$Code)` is ignored when the amount of code in an execution region exceeds 4MB of 16-bit T32 code, 16MB of 32-bit T32 code, and 32MB of A32 code.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

There are no state-change veneers in A64.

Related information

Linker-generated veneers on page 597

3.6.6 Placement of CMSE veneer sections for a Secure image

`armlink` automatically generates all CMSE veneer sections for a Secure image.

The linker:

- Creates `__at` sections that are called `Veneer$$CMSE_AT_<address>` for secure gateway veneers that you specify in a user-defined input import library.
- Produces one normal section `Veneer$$CMSE` to hold all other secure gateway veneers.

Placement of secure gateway veneers generated from input import libraries

The following example shows the placement of secure gateway veneers for functions `entry1` and `entry2` that are specified in the input import library:

```
...

** Section #4 'ER$$Veneer$$CMSE_AT_0x00004000' (SHT_PROGBITS) [SHF_ALLOC +
 SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....    SG        ; [0x3e08]
        0x00004004:    f004b85a    ..Z.    B.W      __acle_se_entry1 ; 0x80bc
    entry2
        0x00004008:    e97fe97f    ....    SG        ; [0x3e10]
        0x0000400c:    f004b868    ..h.    B.W      __acle_se_entry2 ; 0x80e0

...
```

The same rules and options that apply to normal `__at` sections apply to `__at` sections created for secure gateway veneers. The same rules and options also apply to the automatic placement of these sections when you specify `--autoat`.

Placement of secure gateway veneers that are not specified in the input import library

Secure gateway veneers that do not have their addresses specified in an input import library get generated in the `Veneer$$CMSE` input section. You must place this section as required. If you create a simple image, that is without using a scatter file, the sections get placed in the ER_XO execution region, and the respective ER_XO output section.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The following example shows the placement of secure gateway veneers for functions `entry3` and `entry4` that are not specified in the input import library:

```
...

** Section #1 'ER_XO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00008000

    $t
    entry3
        0x00008000:    e97fe97f    ....    SG
        0x00008004:    f000b87e    ..~.    B.W      __acle_se_entry3 ; 0x8104
    entry4
        0x00008008:    e97fe97f    ....    SG
        0x0000800c:    f000b894    ....    B.W      __acle_se_entry4 ; 0x8138

...
```

Placement of secure gateway veneers with a scatter file

To make sure all the secure gateway veneers are in a single section, you must place them using a scatter file.

Secure gateway veneers that are not specified in the input import library are new veneers. New veneers get generated in the `Veneer$$CMSE` input section. You can place this section in the scatter file as required. Veneers that are already present in the input import library are placed at the address that is specified in this library. This placement is done by creating `Veneer$$CMSE_AT_<address>` sections for them. These sections use the same facility that is used by other AT sections. Therefore, if you use `--no_autoat`, you can place these sections either by using the `--autoat` mechanism or by manually placing them using a scatter file.

For a Non-secure callable region of size 0x1000 bytes with a base address of 0x4000 a suitable example of a scatter file load and execution region to match the veneers is:

```
LOAD_NSCR 0x4000 0x1000
{
    EXEC_NSCR 0x4000 0x1000
    {
        *(Veneer$$CMSE)
    }
}
```

The secure gateway veneers are placed as follows:

```
...

** Section #7 'EXEC_NSCR' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR +
 SHF_ARM_NOREAD]
    Size   : 64 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....    SG
        0x00004004:    f7fcb850    ..P.    B        __acle_se_entry1 ; 0xa8
    entry2
        0x00004008:    e97fe97f    ....    SG
        0x0000400c:    f7fcb85e    ..^.    B        __acle_se_entry2 ; 0xcc
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
...

    entry3
        0x00004020:    e97fe97f    ....    SG
        0x00004024:    f7fcb864    ..d.    B        __acle_se_entry3 ; 0xf0
    entry4
        0x00004028:    e97fe97f    ....    SG
        0x0000402c:    f7fcb87a    ..z.    B        __acle_se_entry4 ; 0x124

...
```

Related information

Generation of secure gateway veneers on page 601
Placement of __at sections at a specific address on page 663
Restrictions on placing __at sections on page 664
Automatic placement of __at sections on page 664
Manual placement of __at sections on page 666

3.6.7 Reserving an empty block of memory

You can reserve an empty block of memory with a scatter file, such as the area used for the stack.

To reserve an empty block of memory, add an execution region in the scatter file and assign the `EMPTY` attribute to that region.

3.6.7.1 Characteristics of a reserved empty block of memory

An empty block of memory that is reserved with a scatter-loading description has certain characteristics.

The block of memory does not form part of the load region, but is assigned for use at execution time. Because it is created as a dummy ZI region, the linker uses the following symbols to access it:

- `Image$$<region_name>$$ZI$$Base`.
- `Image$$<region_name>$$ZI$$Limit`.
- `Image$$<region_name>$$ZI$$Length`.

If the length is given as a negative value, the address is taken to be the end address of the region. This address must be an absolute address and not a relative one.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.7.2 Example of reserving an empty block of memory

This example shows how to reserve and empty block of memory for stack and heap using a scatter-loading description. It also shows the related symbols that the linker generates.

In the following example, the execution region definition `STACK 0x800000 EMPTY -0x10000` defines a region that is called `STACK`. The region starts at address `0x7F0000` and ends at address `0x800000`:

```
LR_1 0x80000                            ; load region starts at 0x80000
{
    STACK 0x800000 EMPTY -0x10000       ; region ends at 0x800000 because of the
                                        ; negative length. The start of the region
                                        ; is calculated using the length.
    {
                                        ; Empty region for placing the stack
    }

    HEAP +0 EMPTY 0x10000               ; region starts at the end of previous
                                        ; region. End of region calculated using
                                        ; positive length
    {
                                        ; Empty region for placing the heap
    }
    ...                                 ; rest of scatter-loading description
}
```

Note

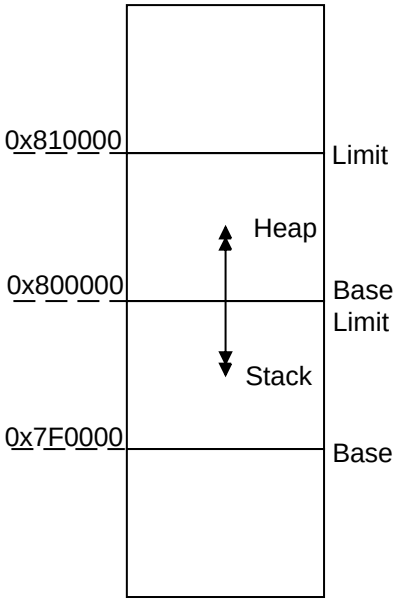
The dummy ZI region that is created for an `EMPTY` execution region is not initialized to zero at runtime.

If the address is in relative (`+<offset>`) form and the length is negative, the linker generates an error.

The following figure shows a diagrammatic representation for this example.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 3-11: Reserving a region for the stack



In this example, the linker generates the following symbols:

```
Image$$STACK$$ZI$$Base      = 0x7f0000
Image$$STACK$$ZI$$Limit     = 0x800000
Image$$STACK$$ZI$$Length    = 0x10000
Image$$HEAP$$ZI$$Base       = 0x800000
Image$$HEAP$$ZI$$Limit      = 0x810000
Image$$HEAP$$ZI$$Length     = 0x10000
```

Note

The `EMPTY` attribute applies only to an execution region. The linker generates a warning and ignores an `EMPTY` attribute that is used in a load region definition.

The linker checks that the address space used for the `EMPTY` region does not overlap any other execution region.

3.6.8 Placement of Arm C and C++ library code

You can place code from the Arm standard C and C++ libraries using a scatter file.

Use `*armlib*` or `*libcxx*` so that the linker can resolve library naming in your scatter file.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Some Arm C and C++ library sections must be placed in a root region, for example `__main.o`, `__scatter*.o`, `__dc*.o`, and `*Region$$Table`. This list can change between releases. The linker can place all these sections automatically in a future-proof way with `InRoot$$Sections`.


Note

For AArch64, `__rtentry*.o` is moved to a root region.

Related information

Region table format

3.6.8.1 Placement of code in a root region

Some code must always be placed in a root region. You do this in a similar way to placing a named section.

To place all sections that must be in a root region, use the section selector `InRoot$$Sections`. For example :

```
ROM_LOAD 0x0000 0x4000
{
  ROM_EXEC 0x0000 0x4000       ; root region at 0x0
  {
    vectors.o (Vect, +FIRST)   ; Vector table
    * (InRoot$$Sections)       ; All library sections that must be in a
                               ; root region, for example, __main.o,
                               ; __scatter*.o, __dc*.o, and *Region$$Table
  }
  RAM 0x10000 0x8000
  {
    * (+RO, +RW, +ZI)          ; all other sections
  }
}
```

Related information

Related information

Region table format

3.6.8.2 Placement of Arm C library code

You can place C library code using a scatter file.

To place C library code, specify the library path and library name as the module selector. You can use wildcard characters if required. For example:

```
LR1 0x0
{
    ROM1 0
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    {
        * (InRoot$$Sections)
        * (+RO)
    }
    ROM2 0x1000
    {
        *armlib/c_* (+RO)                    ; all Arm-supplied C library functions
    }

    RAM1 0x3000
    {
        *armlib* (+RO)                       ; all other Arm-supplied library code
                                             ; for example, floating-point libraries
    }
    RAM2 0x4000
    {
        * (+RW, +ZI)
    }
}
```

The name `armlib` indicates the Arm C library files that are located in the directory `<install_directory>\lib\armlib`.

Related information

Placement of code in a root region on page 691
Placing Arm C++ library code on page 692
C and C++ library naming conventions

3.6.8.3 Placing Arm C++ library code

You can place C++ library code using a scatter file.

About this task

To place C++ library code, specify the library path and library name as the module selector. You can use wildcard characters if required.

Procedure

1. Create the following C++ program, `foo.cpp`:

```
#include <iostream>

using namespace std;

extern "C" int foo ()
{
  cout << "Hello" << endl;
  return 1;
}
```

2. To place the C++ library code, define the following scatter file, `scatter.scat`:

```
LR 0x8000
{
    ER1 +0
    {
        *armlib*(+RO)
    }
    ER2 +0
    {
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
        *libcxx*(+RO)
    }
    ER3 +0
    {
        *(+RO)

        ; All .ARM.exidx* sections must be coalesced into a single contiguous
        ; .ARM.exidx section because the unwinder references linker-generated
        ; Base and Limit symbols for this section.
        *(0x70000001)  ; SHT_ARM_EXIDX sections

        ; All .init_array sections must be coalesced into a single contiguous
        ; .init_array section because the initialization code references
        ; linker-generated Base and Limit for this section.
        *(.init_array)
    }
    ER4 +0
    {
        *(+RW,+ZI)
    }
}
```

The name `*armlib*` matches `<install_directory>\lib\armlib`, indicating the Arm C library files that are located in the `armlib` directory.

The name `*libcxx*` matches `<install_directory>\lib\libcxx`, indicating the C++ library files that are located in the `libcxx` directory.

3. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c foo.cpp
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --scatter=scatter.scat --map main.o foo.o -o foo.axf
```

The `--map` option displays the memory map of the image.

Related information

Placement of code in a root region on page 691
Placement of Arm C library code on page 691
C and C++ library naming conventions

3.6.9 Alignment of regions to page boundaries

You can produce an ELF file with each execution region starting at a page boundary.

The linker provides the following built-in functions to help create load and execution regions on page boundaries:

- `AlignExpr`, to specify an address expression.
- `GetPageSize`, to obtain the page size for use in `AlignExpr`. If you use `GetPageSize`, you must also use the `--paged` linker command-line option.
- `SizeOfHeaders()`, to return the size of the ELF header and Program Header table.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

- Alignment on an execution region causes both the load address and execution address to be aligned.
- The default page size is `0x8000`. To change the page size, specify the `--pagesize` linker command-line option.

To produce an ELF file with each execution region starting on a new page, and with code starting on the next page boundary after the header information:

```
LR1 0x0 + SizeOfHeaders()
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW AlignExpr(+0, GetPageSize())
    {
        *(+RW)
    }
    ER_ZI AlignExpr(+0, GetPageSize())
    {
        *(+ZI)
    }
}
```

If you set up your ELF file in this way, then you can memory-map it onto an operating system in such a way that:

- RO and RW data can be given different memory protections, because they are placed in separate pages.
- The load address everything expects to run at is related to its offset in the ELF file by specifying `SizeOfHeaders()` for the first load region.

Related information

Alignment of execution regions and input sections on page 694
Linker support for creating demand-paged files on page 595
Expression evaluation in scatter files on page 735
Example of using expression evaluation in a scatter file to avoid padding on page 697
Example of aligning a base address in execution space but still tightly packed in load space on page 743
AlignExpr(expr, align) function on page 741
GetPageSize() function on page 742
SizeOfHeaders() function on page 743
Load region attributes on page 715
Execution region attributes on page 723
--paged on page 514
--pagesize=pagesize on page 514

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.10 Alignment of execution regions and input sections

There are situations when you want to align code and data sections. How you deal with them depends on whether you have access to the source code.

Aligning when it is convenient for you to modify the source and recompile

When it is convenient for you to modify the original source code, you can align at compile time with the `__align(n)` keyword, for example.

Aligning when it is not convenient for you to modify the source and recompile

It might not be convenient for you to modify the source code for various reasons. For example, your build process might link the same object file into several images with different alignment requirements.

When it is not convenient for you to modify the source code, then you must use the following alignment specifiers in a scatter file:

`ALIGNALL`

Increases the section alignment of all the sections in an execution region, for example:

```
ER_DATA ... ALIGNALL 8
{
    ... ;selectors
}
```

`OVERALIGN`

Increases the alignment of a specific section, for example:

```
ER_DATA ...
{
    *.o(.bar, OVERALIGN 8)
    ... ;selectors
}
```

Note

`armlink` does not OVERALIGN some sections where it might be unsafe to do so. For more information, see Syntax of an input section description.

Related information

Alignment of regions to page boundaries on page 693
Input section descriptions on page 729
Execution region attributes on page 723

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.11 Preprocessing a scatter file

You can pass a scatter file through a C preprocessor. This permits access to all the features of the C preprocessor.

Use the first line in the scatter file to specify a preprocessor command that the linker invokes to process the file. The command is of the form:

```
#! preprocessor [preprocessor_flags]
```

Most typically the command is of the form `#! armclang --target=<target> -march=<architecture> -E -x c`. This passes the scatter file through the `armclang` preprocessor.

You can:

- Add preprocessing directives to the top of the scatter file.
- Use simple expression evaluation in the scatter file.

For example, a scatter file, `file.scat`, might contain:

```
#! armclang --target=arm-arm-none-eabi -march=armv8-a -E -x c

#define ADDRESS 0x20000000
#include "include_file_1.h"

LR1 ADDRESS
{
    ...
}
```

The linker parses the preprocessed scatter file and treats the directives as comments.

You can also use the `--predefine` command-line option to assign values to constants. For this example:

1. Modify `file.scat` to delete the directive `#define ADDRESS 0x20000000`.
2. Specify the command:

```
armlink --predefine="-DADDRESS=0x20000000" --scatter=file.scat
```

Default behavior for armclang -E in a scatter file

`armlink` behaves in the same way as `armclang` when invoking other Arm tools.

`armlink` searches for the `armclang` binary in the following order:

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

`armlink` invokes `armclang` with the `-I<scatter_file_path>` option so that any preprocessor directives with relative paths work. The linker only adds this option if the full name of the preprocessor tool given is `armclang` or `armclang.exe`. This means that if an absolute path or

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

a relative path is given, the linker does not give the `-I<scatter_file_path>` option to the preprocessor. This also happens with the `--cpu` option.

On Windows, `.exe` suffixes are handled, so `armclang.exe` is considered the same as `armclang`. Executable names are case insensitive, so `armclang` is considered the same as `armclang`. The portable way to write scatter file preprocessing lines is to use correct capitalization and omit the `.exe` suffix.

Use of other preprocessors in a scatter file

You must ensure that the preprocessing command line is appropriate for execution on the host system.

This means:

- The string must be correctly quoted for the host system. The portable way to do this is to use double-quotes.
- Single quotes and escaped characters are not supported and might not function correctly.
- The use of a double-quote character in a path name is not supported and might not work.

These rules also apply to any strings passed with the `--predefine` option.

All preprocessor executables must accept the `-o <file>` option to mean output to file and accept the input as a filename argument on the command line. These options are automatically added to the user command line by `armlink`. Any options to redirect preprocessing output in the user-specified command line are not supported.

3.6.12 Example of using expression evaluation in a scatter file to avoid padding

This example shows how to use expression evaluation in a scatter file to avoid padding.

Using certain scatter-loading attributes in a scatter file can result in a large amount of padding in the image.

To remove the padding caused by the `ALIGN`, `ALIGNALL`, and `FIXED` attributes, use expression evaluation to specify the start address of a load region and execution region. The built-in function `AlignExpr` is available to help you specify address expressions.

Example

The following scatter file produces an image with padding:

```
LR1 0x4000
{
    ER1 +0 ALIGN 0x8000
    {
        ...
    }
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

In this example, the `ALIGN` keyword causes `ER1` to be aligned to a `0x8000` boundary in both the load and the execution view. To align in the load view, the linker must insert `0x4000` bytes of padding.

The following scatter file produces an image without padding:

```
LR1 0x4000
{
    ER1 AlignExpr(+0, 0x8000)
    {
        ...
    }
}
```

Using `AlignExpr` the result of `+0` is aligned to a `0x8000` boundary. This creates an execution region with a load address of `0x4000` but an Execution Address of `0x8000`.

Related information

Example of aligning a base address in execution space but still tightly packed in load space on page 743
AlignExpr(expr, align) function on page 741
Execution region attributes on page 723

3.6.13 Equivalent scatter-loading descriptions for simple images

Although you can use command-line options to scatter-load simple images, you can also use a scatter file.

3.6.13.1 Command-line options for creating simple images

The command-line options `--reloc`, `--ro_base`, `--rw_base`, `--ropi`, `--rwpi`, `--split`, and `--xo_base` create the simple image types.

The simple image types are:

- Type 1 image, one load region and contiguous execution regions.
- Type 2 image, one load region and non-contiguous execution regions.
- Type 3 image, two load regions and non-contiguous execution regions.

You can create the same image types by using the `--scatter` command-line option and a file containing one of the corresponding scatter-loading descriptions.



Note

The option `--reloc` is not supported for AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Type 1 image, one load region and contiguous execution regions on page 699
Load region descriptions on page 713
Type 2 image, one load region and non-contiguous execution regions on page 701
Type 3 image, multiple load regions and non-contiguous execution regions on page 703
--reloc on page 522
--ro_base=address on page 525
--ropi on page 526
--rw_base=address on page 527
--rwpi on page 528
--scatter=filename on page 529
--split on page 538
--xo_base=address on page 560
Load region attributes on page 715

3.6.13.2 Type 1 image, one load region and contiguous execution regions

A Type 1 image consists of a single load region in the load view and up to four execution regions in the execution view. The execution regions are placed contiguously in the memory map.

By default, the ER_RO, ER_RW, and ER_ZI execution regions are present. If an image contains any *eXecute-Only* (XO) sections, then an ER_XO execution region is also present.



Note

XO memory is supported only for Arm®v6-M, Armv7-M, and Armv8-M architectures. However, XO is not supported on Armv6-M for any form of position independent code.

`--ro_base <address>` specifies the load and execution address of the region containing the RO output section. The following example shows the scatter-loading description equivalent to using `--ro_base 0x040000`:

```
LR_1 0x040000     ; Define the load region name as LR_1, the region starts at
 0x040000.
{
    ER_RO +0      ; First execution region is called ER_RO, region starts at end of
                  ; previous region. Because there is no previous region, the
                  ; address is 0x040000.
    {
        * (+RO)   ; All RO sections go into this region, they are placed
                  ; consecutively.
    }
    ER_RW +0      ; Second execution region is called ER_RW, the region starts at
 the
                  ; end of the previous region.
                  ; The address is 0x040000 + size of ER_RO region.
    {
        * (+RW)   ; All RW sections go into this region, they are placed
                  ; consecutively.
    }
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    ER_ZI +0      ; Last execution region is called ER_ZI, the region starts at the
                  ; end of the previous region at 0x040000 + the size of the ER_RO
                  ; regions + the size of the ER_RW regions.
    {
        * (+ZI)   ; All ZI sections are placed consecutively here.
    }
}
```

In this example:

- This description creates an image with one load region called `LR_1` that has a load address of `0x040000`.
- The image has three execution regions, named `ER_RO`, `ER_RW`, and `ER_ZI`, that contain the RO, RW, and ZI output sections respectively. RO and RW are root regions. ZI is created dynamically at runtime. The execution address of `ER_RO` is `0x040000`. All three execution regions are placed contiguously in the memory map by using the `+<offset>` form of the base designator for the execution region description. This enables an execution region to be placed immediately following the end of the preceding execution region.

Use the `--reloc` option to make relocatable images. Used on its own, `--reloc` makes an image similar to simple type 1, but the single load region has the `RELOC` attribute.


Note

The `--reloc` option and `RELOC` attribute are not supported for AArch64 state.

ROPI example variant (AArch32 only)

In this variant, the execution regions are placed contiguously in the memory map. However, `--ropi` marks the load and execution regions containing the RO output section as position independent.

The following example shows the scatter-loading description equivalent to using `--ro_base 0x010000 --ropi`:

```
LR_1 0x010000 PI        ; The first load region is at 0x010000.
{
    ER_RO +0            ; The PI attribute is inherited from parent.
                        ; The default execution address is 0x010000, but the code
                        ; can be moved.
    {
        * (+RO)         ; All the RO sections go here.
    }
    ER_RW +0 ABSOLUTE   ; PI attribute is overridden by ABSOLUTE.
    {
        * (+RW)         ; The RW sections are placed next. They cannot be moved.
    }
    ER_ZI +0            ; ER_ZI region placed after ER_RW region.
    {
        * (+ZI)         ; All the ZI sections are placed consecutively here.
    }
}
```

`ER_RO`, the RO execution region, inherits the `PI` attribute from the load region `LR_1`. The next execution region, `ER_RW`, is marked as `ABSOLUTE` and uses the `+<offset>` form of base designator.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

This prevents `ER_RW` from inheriting the `PI` attribute from `ER_RO`. Also, because the `ER_ZI` region has an offset of `+0`, it inherits the `ABSOLUTE` attribute from the `ER_RW` region.


Note

If an image contains execute-only sections, ROPI is not supported. If you use `--ropi` to link such an image, `armlink` gives an error.

Related information

Command-line options for creating simple images on page 698
Load region descriptions on page 713
Considerations when using a relative address +offset for a load region on page 719
Considerations when using a relative address +offset for execution regions on page 729
--ro_base=address on page 525
--ropi on page 526
Load region attributes on page 715
--reloc on page 522

3.6.13.3 Type 2 image, one load region and non-contiguous execution regions

A Type 2 image consists of a single load region in the load view and three execution regions in the execution view. It is similar to images of Type 1 except that the RW execution region is not contiguous with the RO execution region.

`--ro_base=<address>` specifies the load and execution address of the region containing the RO output section. `--rw_base=<address>` specifies the execution address for the RW execution region.

For images that contain *eXecute-Only* (XO) sections, the XO execution region is placed at the address specified by `--ro_base`. The RO execution region is placed contiguously and immediately after the XO execution region.

If you use `--xo_base <address>`, then the XO execution region is placed in a separate load region at the specified address.


Note

XO memory is supported only for Arm®v6-M, Armv7-M, and Armv8-M architectures. However, XO is not supported on Armv6-M for any form of position independent code.

Example for single load region and multiple execution regions

The following example shows the scatter-loading description equivalent to using `--ro_base=0x010000 --rw_base=0x040000`:

```
LR_1 0x010000        ; Defines the load region name as LR_1
{
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    ER_RO +0          ; The first execution region is called ER_RO and starts at end
                      ; of previous region. Because there is no previous region, the
                      ; address is 0x010000.
    {
        * (+RO)       ; All RO sections are placed consecutively into this region.
    }
    ER_RW 0x040000    ; Second execution region is called ER_RW and starts at
 0x040000.
    {
        * (+RW)       ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0          ; The last execution region is called ER_ZI.
                      ; The address is 0x040000 + size of ER_RW region.
    {
        * (+ZI)       ; All ZI sections are placed consecutively here.
    }
}
```

In this example:

- This description creates an image with one load region, named `LR_1`, with a load address of `0x010000`.
- The image has three execution regions, named `ER_RO`, `ER_RW`, and `ER_ZI`, that contain the RO, RW, and ZI output sections respectively. The RO region is a root region. The execution address of `ER_RO` is `0x010000`.
- The `ER_RW` execution region is not contiguous with `ER_RO`. Its execution address is `0x040000`.
- The `ER_ZI` execution region is placed immediately following the end of the preceding execution region, `ER_RW`.

RWPI example variant (AArch32 only)

This is similar to images of Type 2 with `--rw_base` where the RW execution region is separate from the RO execution region. However, `--rwpi` marks the execution regions containing the RW output section as position independent.

The following example shows the scatter-loading description equivalent to using `--ro_base=0x010000 --rw_base=0x018000 --rwpi`:

```
LR_1 0x010000           ; The first load region is at 0x010000.
{
    ER_RO +0            ; Default ABSOLUTE attribute is inherited from parent.
                        ; The execution address is 0x010000. The code and RO data
                        ; cannot be moved.
    {
        * (+RO)         ; All the RO sections go here.
    }
    ER_RW 0x018000 PI   ; PI attribute overrides ABSOLUTE
    {
        * (+RW)         ; The RW sections are placed at 0x018000 and they can be
                        ; moved.
    }
    ER_ZI +0            ; ER_ZI region placed after ER_RW region.
    {
        * (+ZI)         ; All the ZI sections are placed consecutively here.
    }
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

ER_RO, the RO execution region, inherits the ABSOLUTE attribute from the load region LR_1. The next execution region, ER_RW, is marked as PI. Also, because the ER_ZI region has an offset of +0, it inherits the PI attribute from the ER_RW region.

Similar scatter-loading descriptions can also be written to correspond to the usage of other combinations of --ropi and --rwpi with Type 2 and Type 3 images.

Related information

Load region descriptions on page 713
Considerations when using a relative address +offset for a load region on page 719
Considerations when using a relative address +offset for execution regions on page 729
--ro_base=address on page 525
--rw_base=address on page 527
--xo_base=address on page 560
Load region attributes on page 715

3.6.13.4 Type 3 image, multiple load regions and non-contiguous execution regions

A Type 3 image consists of multiple load regions in load view and multiple execution regions in execution view. They are similar to images of Type 2 except that the single load region in Type 2 is now split into multiple load regions.

You can relocate and split load regions using the following linker options:

--reloc
The combination --reloc --split makes an image similar to simple Type 3, but the two load regions now have the RELOC attribute.

--ro_base=<address1>
Specifies the load and execution address of the region containing the RO output section.

--rw_base=<address2>
Specifies the load and execution address for the region containing the RW output section.

--xo_base=<address3>
Specifies the load and execution address for the region containing the *eXecute-Only* (XO) output section, if present.

--split
Splits the default single load region that contains the RO and RW output sections into two load regions. One load region contains the RO output section and one contains the RW output section.


Note

For images containing XO sections, and if --xo_base is not used, an XO execution region is placed at the address specified by --ro_base. The RO execution region is placed immediately after the XO region.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

XO memory is supported only for Arm®v6-M, Armv7-M, and Armv8-M architectures. However, XO is not supported on Armv6-M for any form of position independent code.

Example for multiple load regions

The following example shows the scatter-loading description equivalent to using `--ro_base=0x010000 --rw_base=0x040000 --split`:

```
LR_1 0x010000     ; The first load region is at 0x010000.
{
    ER_RO +0      ; The address is 0x010000.
    {
        * (+RO)
    }
}
LR_2 0x040000     ; The second load region is at 0x040000.
{
    ER_RW +0      ; The address is 0x040000.
    {
        * (+RW)   ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0      ; The address is 0x040000 + size of ER_RW region.
    {
        * (+ZI)   ; All ZI sections are placed consecutively into this region.
    }
}
```

In this example:

- This description creates an image with two load regions, named `LR_1` and `LR_2`, that have load addresses `0x010000` and `0x040000`.
- The image has three execution regions, named `ER_RO`, `ER_RW` and `ER_ZI`, that contain the RO, RW, and ZI output sections respectively. The execution address of `ER_RO` is `0x010000`.
- The `ER_RW` execution region is not contiguous with `ER_RO`, because its execution address is `0x040000`.
- The `ER_ZI` execution region is placed immediately after `ER_RW`.

Example for multiple load regions with an XO region

The following example shows the scatter-loading description equivalent to using `--ro_base=0x010000 --rw_base=0x040000 --split` when an object file has XO sections:

```
LR_1 0x010000     ; The first load region is at 0x010000.
{
    ER_XO +0      ; The address is 0x010000.
    {
        * (+XO)
    }
    ER_RO +0      ; The address is 0x010000 + size of ER_XO region.
    {
        * (+RO)
    }
}
LR_2 0x040000     ; The second load region is at 0x040000.
{
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    ER_RW +0      ; The address is 0x040000.
    {
        * (+RW)   ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0      ; The address is 0x040000 + size of ER_RW region.
    {
        * (+ZI)   ; All ZI sections are placed consecutively into this region.
    }
}
```

In this example:

- This description creates an image with two load regions, named `LR_1` and `LR_2`, that have load addresses `0x010000` and `0x040000`.
- The image has four execution regions, named `ER_XO`, `ER_RO`, `ER_RW` and `ER_ZI`, that contain the XO, RO, RW, and ZI output sections respectively. The execution address of `ER_XO` is placed at the address specified by `--ro_base`, `0x010000`. `ER_RO` is placed immediately after `ER_XO`.
- The `ER_RW` execution region is not contiguous with `ER_RO`, because its execution address is `0x040000`.
- The `ER_ZI` execution region is placed immediately after `ER_RW`.



Note

If you also specify `--xo_base`, then the `ER_XO` execution region is placed in a load region separate from the `ER_RO` execution region, at the specified address.

Relocatable load regions example variant

This Type 3 image also consists of two load regions in load view and three execution regions in execution view. However, `--reloc` specifies that the two load regions now have the `RELOC` attribute.

The following example shows the scatter-loading description equivalent to using `--ro_base 0x010000 --rw_base 0x040000 --reloc --split`:

```
LR_1 0x010000 RELOC
{
     ER_RO + 0
     {
         * (+RO)
     }
}
LR2 0x040000 RELOC
{
     ER_RW + 0
     {
         * (+RW)
     }
     ER_ZI +0
     {
         * (+ZI)
     }
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Load region descriptions on page 713
Considerations when using a relative address +offset for a load region on page 719
Considerations when using a relative address +offset for execution regions on page 729
--reloc on page 522
--ro_base=address on page 525
--rw_base=address on page 527
--split on page 538
--xo_base=address on page 560
Load region attributes on page 715
Inheritance rules for load region address attributes on page 717
Inheritance rules for the RELOC address attribute on page 719
Inheritance rules for execution region address attributes on page 727

3.6.14 How the linker resolves multiple matches when processing scatter files

An input section must be unique. In the case of multiple matches, the linker attempts to assign the input section to a region based on the attributes of the input section description.

The linker assignment of the input section is based on a `<module_select_pattern>` and `<input_section_selector>` pair that is the most specific. However, if a unique match cannot be found, the linker faults the scatter-loading description.

The following variables describe how the linker matches multiple input sections:

- `<m1>` and `<m2>` represent module selector patterns.
- `<s1>` and `<s2>` represent input section selectors.

For example, if input section A matches `<m1,s1>` for execution region R1, and A matches `<m2,s2>` for execution region R2, the linker:

- Assigns A to R1 if `<m1,s1>` is more specific than `<m2,s2>`.
- Assigns A to R2 if `<m2,s2>` is more specific than `<m1,s1>`.
- Diagnoses the scatter-loading description as faulty if `<m1,s1>` is not more specific than `<m2,s2>` and `<m2,s2>` is not more specific than `<m1,s1>`.

`armlink` uses the following strategy to determine the most specific `<module_select_pattern>`, `<input_section_selector>` pair:

Resolving the priority of two module_selector, section_selector pairs m1, s1 and m2, s2

The strategy starts with two `<module_select_pattern, input_section_selector>` pairs. `<m1,s1>` is more specific than `<m2,s2>` only if any of the following are true:

1. `<s1>` is either a literal input section name, that is it contains no pattern characters, or a section type and `<s2>` matches input section attributes.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2. <m1> is more specific than <m2>.
3. <s1> is more specific than <s2>.

The conditions are tested in order so condition 1 takes precedence over condition 2 and 3, and condition 2 takes precedence over condition 3.

Resolving the priority of two module selectors m1 and m2 in isolation

For the module selector patterns, <m1> is more specific than <m2> if the text string <m1> matches pattern <m2> and the text string <m2> does not match pattern <m1>.

Resolving the priority of two section selectors s1 and s2 in isolation

For the input section selectors:

- If one of <s1> or <s2> matches the input section name or type and the other matches the input section attributes, <s1> and <s2> are unordered and the description is diagnosed as faulty. For example, using `*(+RO)` gives ambiguous matches with `*(.rodata*)`.
- If both <s1> and <s2> match the input section name or type, the following relationships determine whether <s1> is more specific than <s2>:
  - Section type is more specific than section name.
  - If both <s1> and <s2> match input section type, <s1> and <s2> are unordered and the description is diagnosed as faulty.
  - If <s1> and <s2> are both patterns matching section names, the same definition as for module selector patterns is used.
- If both <s1> and <s2> match input section attributes, the following relationships determine whether <s1> is more specific than <s2>:
  - `ENTRY` is more specific than `RO-CODE`, `RO-DATA`, `RW-CODE`, or `RW-DATA`.
  - `RO-CODE` is more specific than `RO`.
  - `RO-DATA` is more specific than `RO`.
  - `RW-CODE` is more specific than `RW`.
  - `RW-DATA` is more specific than `RW`.
  - There are no other members of the (<s1> more specific than <s2>) relationship between section attributes.

This matching strategy has the following consequences:

- Descriptions do not depend on the order they are written in the file.
- Generally, the more specific the description of an object, the more specific the description of the input sections it contains.
- The <input_section_selector>s are not examined unless:
  - Object selection is inconclusive.
  - One selector specifies a literal input section name or a section type and the other selects by attribute. In this case, the explicit input section name or type is more specific than any attribute. This is true even if the object selector associated with the input section name is less specific than that of the attribute.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The `.ANY` module selector is available to assign any sections that cannot be resolved from the scatter-loading description.

Example

The following example shows multiple execution regions and pattern matching:

```
LR_1 0x040000
{
    ER_ROM 0x040000              ; The startup exec region address is the same
    {                            ; as the load address.
        application.o (+ENTRY)   ; The section containing the entry point from
    }                            ; the object is placed here.
    ER_RAM1 0x048000
    {
        application.o (+RO-CODE) ; Other RO code from the object goes here
    }
    ER_RAM2 0x050000
    {
        application.o (+RO-DATA) ; The RO data goes here
    }
    ER_RAM3 0x060000
    {
        application.o (+RW)      ; RW code and data go here
    }
    ER_RAM4 +0                   ; Follows on from end of ER_R3
    {
        *.o (+RO, +RW, +ZI)      ; Everything except for application.o goes here
    }
}
```

Related information

Manual placement of unassigned sections on page 670
Input section descriptions on page 729
Syntax of a scatter file on page 711
Syntax of an input section description on page 730

3.6.15 How the linker resolves path names when processing scatter files

The linker matches wildcard patterns in scatter files against any combination of forward slashes and backslashes it finds in path names.

This might be useful where the paths are taken from environment variables or multiple sources, or where you want to use the same scatter file to build on Windows or Unix platforms.



Note

Use forward slashes in path names to ensure they are understood on Windows and Unix platforms.

Related information

Syntax of a scatter file on page 711

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.6.16 Scatter file to ELF mapping

Shows how scatter file components map onto ELF.

ELF executable files contain segments:

- A load region is represented by an ELF program segment with type PT_LOAD.
- An execution region is represented by one or more of the following ELF sections:
  - XO.
  - RO.
  - RW.
  - ZI.



Note

If XO and RO are mixed within an execution region, that execution region is treated as RO.

For example, you might have a scatter file similar to the following:

```
LOAD 0x8000
{
    EXEC_ROM +0
    {
        *(+RO)
    }
    RAM +0
    {
        *(+RW,+ZI)
    }
    HEAP +0x100 EMPTY 0x100
    {
    }
    STACK +0 EMPTY 0x400
    {
    }
}
```

This scatter file creates a single program segment with type PT_LOAD for the load region with address `0x8000`.

A single output section with type SHT_PROGBITS is created to represent the contents of EXEC_ROM. Two output sections are created to represent RAM. The first has a type SHT_PROGBITS and contains the initialized read/write data. The second has a type of SHT_NOBITS and describes the zero-initialized data.

The heap and stack are described in the ELF file by SHT_NOBITS sections.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Enter the following `fromelf` command to see the scatter-loaded sections in the image:

```
fromelf --text -v my_image.axf
```

To display the symbol table, enter the command:

```
fromelf --text -s -v my_image.axf
```

The following is an example of the `fromelf` output showing the `LOAD`, `EXEC_ROM`, `RAM`, `HEAP`, and `STACK` sections:

```
...
============================================================================
** Program header #0
    Type          : PT_LOAD (1)
    File Offset   : 52 (0x34)
    Virtual Addr  : 0x00008000
    Physical Addr : 0x00008000
    Size in file  : 764 bytes (0x2fc)
    Size in memory: 2140 bytes (0x85c)
    Flags         : PF_X + PF_W + PF_R + PF_ARM_ENTRY (0x80000007)
    Alignment     : 4
============================================================================
** Section #1
    Name        : EXEC_ROM
...
    Addr        : 0x00008000
    File Offset : 52 (0x34)
    Size        : 740 bytes (0x2e4)
...
====================================
** Section #2
    Name        : RAM
...
    Addr        : 0x000082e4
    File Offset : 792 (0x318)
    Size        : 20 bytes (0x14)
...
====================================
** Section #3
    Name        : RAM
...
    Addr        : 0x000082f8
    File Offset : 812 (0x32c)
    Size        : 96 bytes (0x60)
...
====================================
** Section #4
    Name        : HEAP
...
    Addr        : 0x00008458
    File Offset : 812 (0x32c)
    Size        : 256 bytes (0x100)
...
====================================
** Section #5
    Name        : STACK
...
    Addr        : 0x00008558
    File Offset : 812 (0x32c)
    Size        : 1024 bytes (0x400)
...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Overview of scatter-loading on page 643
Scatter-loading images with a simple memory map on page 648

3.7 Scatter File Syntax

Describes the format of scatter files.

3.7.1 BNF notation used in scatter-loading description syntax

Scatter-loading description syntax uses standard *Backus-Naur Form* (BNF) notation.

The following table summarizes the BNF symbols that are used for describing the syntax of scatter-loading descriptions.

Table 3-21: BNF notation

Symbol	Description
"	Quotation marks indicate that a character that is normally part of the BNF syntax is used as a literal character in the definition. The definition `B"+"C`, for example, can only be replaced by the pattern `B+C`. The definition `B+C` can be replaced by, for example, patterns `BC`, `BBC`, or `BBBC`.
`<A>::=<B>`	Defines <A> as <B>. For example, `A::= B"+" | C` means that <A> is equivalent to either B+ or C. The ::= notation defines a higher level construct in terms of its components. Each component might also have a ::= definition that defines it in terms of even simpler components. For example, `A::= B` and `B::= C | D` means that the definition <A> is equivalent to the patterns `C` or `D`.
`[<A>]`	Optional element <A>. For example, `A::= B[C]D` means that the definition <A> can be expanded into either `BD` or `BCD`.
`<A>+`	Element <A> can have one or more occurrences. For example, `A::= B+` means that the definition <A> can be expanded into `B`, `BB`, or `BBB`.
`<A>*`	Element <A> can have zero or more occurrences.
`<A> | <B>`	Either element <A> or <B> can occur, but not both.
`(<A> <B>)`	Element <A> and <B> are grouped together. This is particularly useful when the | operator is used or when a complex pattern is repeated. For example, `A::=(B C)+ (D | E)` means that the definition <A> can be expanded into any of `BCD`, `BCE`, `BCBCD`, `BCBCE`, `BCBCBCD`, or `BCBCBCE`.

Related information

Syntax of a scatter file on page 711

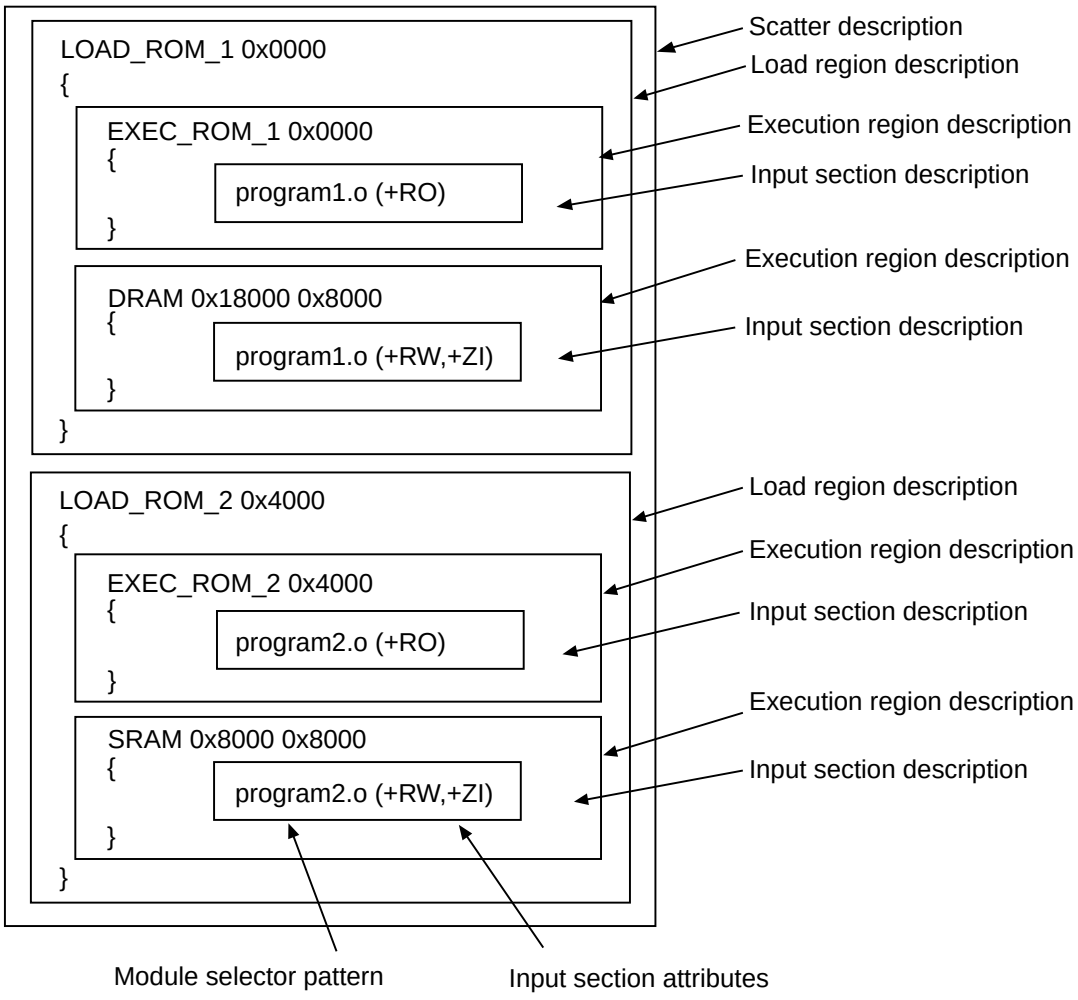
Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.7.2 Syntax of a scatter file

A scatter file contains one or more load regions. Each load region can contain one or more execution regions.

The following figure shows the components and organization of a typical scatter file:

Figure 3-12: Components of a scatter file



Related information

Load region descriptions on page 713
Execution region descriptions on page 720
Scatter-loading Features on page 643

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.7.3 Load region descriptions

A load region description specifies the region of memory where its child execution regions are to be placed.

Related information

Components of a load region description on page 713
Syntax of a load region description on page 714
Load region attributes on page 715
Inheritance rules for load region address attributes on page 717
Inheritance rules for the RELOC address attribute on page 719
Considerations when using a relative address +offset for a load region on page 719

3.7.3.1 Components of a load region description

The components of a load region description allow you to uniquely identify a load region and to control what parts of an ELF file are placed in that region.

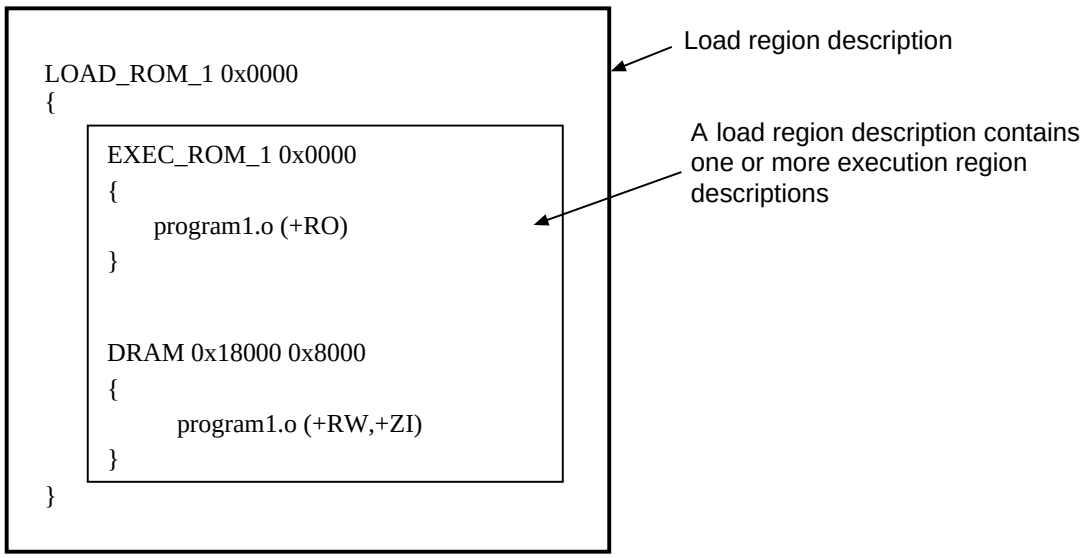
A load region description has the following components:

- A name that is used by the linker to identify different load regions.
- A base address that specifies the start address for the code and data in the load view.
- Attributes that specify the properties of the load region.
- An optional maximum size specification.
- One or more execution regions.

The following figure shows an example of a typical load region description:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 3-13: Components of a load region description



Related information

Syntax of a load region description on page 714
Load region attributes on page 715
Inheritance rules for load region address attributes on page 717
Inheritance rules for the RELOC address attribute on page 719
Inheritance rules for execution region address attributes on page 727
Alignment of regions to page boundaries on page 693
Scatter-loading Features on page 643
Expression evaluation in scatter files on page 735

3.7.3.2 Syntax of a load region description

A load region can contain one or more execution region descriptions.

The syntax of a load region description, in *Backus-Naur Form* (BNF), is:

```
<load_region_description> ::=
  <load_region_name> (<base_address> | ("+" <offset>)) [<attribute_list>]
 [<max_size>]
       "{"
          <execution_region_description>+
       "}"
```

Where:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<load_region_name>

Names the load region. You can use a quoted name. The name is case-sensitive only if you use any region-related linker-defined symbols.

<base_address>

Specifies the address where objects in the region are to be linked. <base_address> must satisfy the alignment constraints of the load region.

+<offset>

Describes a base address that is <offset> bytes beyond the end of the preceding load region. The value of <offset> must be zero modulo four. If this is the first load region, then +<offset> means that the base address begins <offset> bytes from zero.

If you use +<offset>, then the load region might inherit certain attributes from a previous load region.

<attribute_list>

The attributes that specify the properties of the load region contents.

<max_size>

Specifies the maximum size of the load region. This is the size of the load region before any decompression or zero initialization take place. If the optional <max_size> value is specified, armlink generates an error if the region has more than <max_size> bytes allocated to it.

<execution_region_description>

Specifies the execution region name, address, and contents.

Note

The BNF definitions contain additional line returns and spaces to improve readability. They are not required in scatter-loading descriptions and are ignored if present in a scatter file.

Related information

Components of a load region description on page 713
Load region attributes on page 715
Inheritance rules for the RELOC address attribute on page 719
BNF notation used in scatter-loading description syntax on page 711
Considerations when using a relative address +offset for a load region on page 719
Inheritance rules for load region address attributes on page 717
Syntax of a scatter file on page 711
Expression evaluation in scatter files on page 735
Region-related symbols on page 624

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.7.3.3 Load region attributes

A load region has attributes that allow you to control where parts of your image are loaded in the target memory.

The load region attributes are:

ABSOLUTE

The content is placed at a fixed address that does not change after linking. The load address of the region is specified by the base designator. This is the default, unless you use `PI` or `RELOC`.

ALIGN <alignment>

Changes the alignment to the maximum of all input sections and the `ALIGN` value. The minimum section alignment is 4, 8, or 16 based on the minimum alignment requirement of the created data or the user-defined alignment.

`<alignment>` must be a positive power of 2. If the load region has a `<base_address>` then this must be `<alignment>` aligned. If the load region has a `+<offset>` then the linker aligns the calculated base address of the region to an `<alignment>` boundary.

This can also affect the offset in the ELF file. For example, the following causes the data for `FOO` to be written out at 4k offset into the ELF file:

```
FOO +4 ALIGN 4096
```

NOCOMPRESS

RW data compression is enabled by default. The `NOCOMPRESS` keyword enables you to specify that the contents of a load region must not be compressed in the final image.

OVERLAY

The `OVERLAY` keyword enables you to have multiple load regions at the same address. Arm tools do not provide an overlay mechanism. To use multiple load regions at the same address, you must provide your own overlay manager.

The content is placed at a fixed address that does not change after linking. The content might overlap with other regions designated as `OVERLAY` regions.

This attribute modifies the behavior of the `--merge` and `--merge_litpools` command-line options.

PI

This region is position independent. The content does not depend on any fixed address and might be moved after linking without any extra processing. This attribute is optional for AArch32 state, and if used, enables extra dignostic messages.


Note

This attribute is not supported for the following cases:

- If an image contains execute-only sections.
- Mixing absolute linked and bare-metal PIE images. You must link them as two separate units.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- AArch64 state.
- When linking with `--sysv` or `--shared`.

This attribute is supported only for *Read-Only Position Independent* (ROPI) and *Read/Write Position Independent* (RWPI) for AArch32 state. It is not supported for .

PROTECTED

The `PROTECTED` keyword prevents:

- Overlapping of load regions.
- Veneer sharing.
- Constant string sharing with the `--merge` option.
- Constant sharing with the `--merge_litpools` option.

RELOC

`RELOC` is not supported for AArch64 state.

This region is relocatable. The content depends on fixed addresses. Relocation information is output to enable the content to be moved to another location by another tool.

Related information

--merge, --no_merge on page 507
--merge_litpools, --no_merge_litpools on page 508
Components of a load region description on page 713
Syntax of a load region description on page 714
Example of aligning a base address in execution space but still tightly packed in load space on page 743
Section alignment with the linker on page 594
Reuse of veneers when scatter-loading on page 600
Alignment of regions to page boundaries on page 693
Considerations when using a relative address +offset for a load region on page 719
Inheritance rules for load region address attributes on page 717
Inheritance rules for the RELOC address attribute on page 719
Veneer sharing on page 598
Generation of position independent to absolute veneers on page 600
Optimization with RW data compression on page 612
Interaction of OVERLAY and PROTECTED attributes with armlink merge options

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.7.3.4 Inheritance rules for load region address attributes

A load region can inherit the attributes of a previous load region.

For a load region to inherit the attributes of a previous load region, specify a `+<offset>` base address for that region. A load region cannot inherit attributes if:

- You explicitly set the attribute of that load region.
- The load region immediately before has the `OVERLAY` attribute.

You can explicitly set a load region with the `ABSOLUTE`, `PI`, `RELOC`, or `OVERLAY` address attributes.

Note

`PI` and `RELOC` are not supported for AArch64 state.

The following inheritance rules apply when no address attribute is specified:

- The `OVERLAY` attribute cannot be inherited. A region with the `OVERLAY` attribute cannot inherit.
- A base address load or execution region always defaults to `ABSOLUTE`.
- A `+<offset>` load region inherits the address attribute from the previous load region or `ABSOLUTE` if no previous load region exists.

Example

This example shows the inheritance rules for setting the address attributes of load regions:

```
LR1 0x8000 PI
{
    ...
}
LR2 +0              ; LR2 inherits PI from LR1
{
    ...
}
LR3 0x1000          ; LR3 does not inherit because it has no relative base
                      address, gets default of ABSOLUTE
{
    ...
}
LR4 +0              ; LR4 inherits ABSOLUTE from LR3
{
    ...
}
LR5 +0 RELOC        ; LR5 does not inherit because it explicitly sets RELOC
{
    ...
}
LR6 +0 OVERLAY      ; LR6 does not inherit, an OVERLAY cannot inherit
{
    ...
}
LR7 +0              ; LR7 cannot inherit OVERLAY, gets default of ABSOLUTE
{
    ...
}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Components of a load region description on page 713
Components of an execution region description on page 720
Inheritance rules for execution region address attributes on page 727

3.7.3.5 Inheritance rules for the RELOC address attribute

You can explicitly set the `RELOC` attribute for a load region. However, an execution region can only inherit the `RELOC` attribute from the parent load region.



Note

`RELOC` is not supported for AArch64 state.

Example

This example shows the inheritance rules for setting the address attributes with `RELOC`:

```
LR1 0x8000 RELOC
{
    ER1 +0 ; inherits RELOC from LR1
    {
        ...
    }
    ER2 +0 ; inherits RELOC from ER1
    {
        ...
    }
    ER3 +0 RELOC ; Error cannot explicitly set RELOC on an execution region
    {
        ...
    }
}
```

Related information

Components of a load region description on page 713
Syntax of a load region description on page 714
Components of an execution region description on page 720
Inheritance rules for load region address attributes on page 717
Inheritance rules for execution region address attributes on page 727
Considerations when using a relative address +offset for execution regions on page 729
Considerations when using a relative address +offset for a load region on page 719

3.7.3.6 Considerations when using a relative address +offset for a load region

There are some considerations to be aware of when using a relative address for a load region.

When using `+<offset>` to specify a load region base address:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- If the `+<offset>` load region LR2 follows a load region LR1 containing ZI data, then LR2 overlaps the ZI data. To fix this overlap, use the `ImageLimit()` function to specify the base address of LR2. See Scatter files containing relative base address load regions and a ZI execution region for an example.
- A `+<offset>` load region LR2 inherits the attributes of the load region LR1 immediately before it, unless:
  - LR1 has the `OVERLAY` attribute.
  - LR2 has an explicit attribute set.

  If a load region is unable to inherit an attribute, then it gets the attribute `ABSOLUTE`.

  See Inheritance rules for load region address attributes for an example.
- A gap might exist in a ROM image between a `+<offset>` load region and a preceding region when the preceding region has RW data compression applied. This gap appears because the linker calculates the `+<offset>` based on the uncompressed size of the preceding region. However, the gap disappears when the RW data is decompressed at load time.

Related information

Inheritance rules for load region address attributes on page 717
Execution address built-in functions for use in scatter files on page 737
Scatter files containing relative base address load regions and a ZI execution region on page 744

3.7.4 Execution region descriptions

An execution region description specifies the region of memory where parts of your image are to be placed at run-time.

Related information

Components of an execution region description on page 720
Syntax of an execution region description on page 721
Execution region attributes on page 723
Inheritance rules for execution region address attributes on page 727
Considerations when using a relative address +offset for execution regions on page 729

3.7.4.1 Components of an execution region description

The components of an execution region description allow you to uniquely identify each execution region and its position in the parent load region, and to control what parts of an ELF file are placed in that execution region.

An execution region description has the following components:

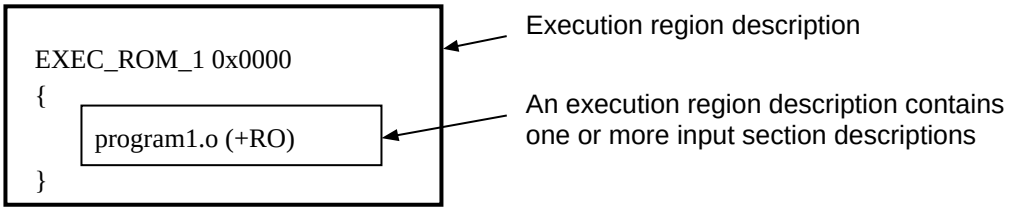
- A name that is used by the linker to identify different execution regions.
- A base address that is either absolute or relative.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Attributes that specify the properties of the execution region.
- An optional maximum size specification.
- One or more Input section descriptions that specify the modules that are placed into this execution region.

The following figure shows the components of a typical execution region description:

Figure 3-14: Components of an execution region description



Related information

Inheritance rules for load region address attributes on page 717
Inheritance rules for the RELOC address attribute on page 719
Inheritance rules for execution region address attributes on page 727
Syntax of an execution region description on page 721
Execution region attributes on page 723

3.7.4.2 Syntax of an execution region description

An execution region specifies where the input sections are to be placed in target memory at run-time.

The syntax of an execution region description, in *Backus-Naur Form* (BNF), is:

```
<execution_region_description> ::=
  <exec_region_name> (<base_address> | "+" <offset>) [<attribute_list>] [<max_size>
 | <length>]
        "{"
            <input_section_description>*
        "}"
```

Where:

<exec_region_name>

Names the execution region. You can use a quoted name. The name is case-sensitive only if you use any region-related linker-defined symbols.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<base_address>

Specifies the address where objects in the region are to be linked. <base_address> must be word-aligned.



Using ALIGN on an execution region causes both the load address and execution address to be aligned.

+<offset>

Describes a base address that is +<offset> bytes beyond the end of the preceding execution region. The value of +<offset> must be zero modulo four.

If this is the first execution region in the load region then +<offset> means that the base address begins +<offset> bytes after the base of the containing load region.

If you use +<offset>, then the execution region might inherit certain attributes from the parent load region, or from a previous execution region within the same load region.

<attribute_list>

The attributes that specify the properties of the execution region contents.

<max_size>

For an execution region marked EMPTY or FILL the <max_size> value is interpreted as the length of the region. Otherwise the <max_size> value is interpreted as the maximum size of the execution region.

[-]<length>

Can only be used with EMPTY to represent a stack that grows down in memory. If the length is given as a negative value, the <base_address> is taken to be the end address of the region.

<input_section_description>

Specifies the content of the input sections.



The BNF definitions contain additional line returns and spaces to improve readability. They are not required in scatter-loading descriptions and are ignored if present in a scatter file.

Related information

Components of an execution region description on page 720
Execution region attributes on page 723
Scatter-loading Features on page 643
Considerations when using a relative address +offset for execution regions on page 729
Expression evaluation in scatter files on page 735
Region-related symbols on page 624
Alignment of regions to page boundaries on page 693

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Inheritance rules for load region address attributes on page 717
Inheritance rules for the RELOC address attribute on page 719
Input section descriptions on page 729

3.7.4.3 Execution region attributes

An execution region has attributes that allow you to control where parts of your image are loaded in the target memory at runtime.

The execution region attributes are:

ABSOLUTE

The content is placed at a fixed address that does not change after linking. A base designator specifies the execution address of the region.

ALIGN <alignment>

Increase the alignment constraint for the execution region from 4 to `<alignment>`. `<alignment>` must be a positive power of 2. If the execution region has a `<base_address>`, then the address must be `<alignment>` aligned. If the execution region has a `+<offset>`, then the linker aligns the calculated base address of the region to an `<alignment>` boundary.



`ALIGN` on an execution region causes both the load address and execution address to be aligned. This alignment can result in padding being added to the ELF file. To align only the execution address, use the `AlignExpr` expression on the base address.

ALIGNALL <value>

Increases the alignment of sections within the execution region.

The value must be a positive power of 2 and must be greater than or equal to 4.

ANY_SIZE <max_size>

Specifies the maximum size within the execution region that `armlink` can fill with unassigned sections. You can use a simple expression to specify the `<max_size>`. That is, you cannot use functions such as `ImageLimit()`.



Specifying `ANY_SIZE` overrides any effects that `--any_contingency` has on the region.

Be aware of the following restrictions when using this keyword:

- `<max_size>` must be less than or equal to the region size.
- You can use `ANY_SIZE` on a region without a `.ANY` selector but `armlink` ignores it.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

AUTO_OVERLAY

Use to indicate regions of memory where `armlink` assigns the overlay sections for loading into at runtime. Overlay sections are those named `.ARM.overlay<N>` in the input object.

The execution region must not have any section selectors.

The addresses that you give for the execution regions are the addresses that `armlink` expects the overlaid code to be loaded at when running. The load region containing the execution regions is where `armlink` places the overlay contents.

By default, the overlay manager loads overlays by copying them into RAM from some other memory that is not suitable for direct execution. For example, very slow Flash or memory from which instruction fetches are not enabled. You can keep your unloaded overlays in peripheral storage that is not mapped into the address space of the processor. To keep such overlays in peripheral storage, you must extract the data manually from the linked image.

`armlink` allocates every overlay to one of the `AUTO_OVERLAY` execution regions, and has to be loaded into only that region to run correctly.

You must use the `--overlay_veneers` command-line option when linking with a scatter file containing the `AUTO_OVERLAY` attribute.


Note

With the `AUTO_OVERLAY` attribute, `armlink` decides how your code sections get allocated to overlay regions. With the `OVERLAY` attribute, you must manually arrange the allocation of the code sections.


Note

Arm® Compiler for Embedded does not support using both manual and automatic overlays within the same program.

EMPTY [-]<length>

Reserves an empty block of memory of a given size in the execution region, typically used by a heap or stack. No section selectors can be placed in a region with the `EMPTY` attribute.

`<length>` represents a stack that grows down in memory. If the length is given as a negative value, the `<base_address>` is taken to be the end address of the region.

FILL <value>

Creates a linker-generated region containing a `<value>`. If you specify `FILL`, you must give a value, for example: `FILL 0xFFFFFFFF`. The `FILL` attribute replaces the following combination: `EMPTY ZEROPAD PADVALUE`.

In certain situations, such as a simulation, filling a region with a value is preferable to spending a long time in a zeroing loop.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

FIXED

Fixed address. The linker attempts to make the execution address equal the load address. If it succeeds, then the region is a root region. If it does not succeed, then the linker produces an error.


Note

The linker inserts padding with this attribute.

NOCOMPRESS

RW data compression is enabled by default. The `NOCOMPRESS` keyword enables you to specify that RW data in an execution region must not be compressed in the final image.

OVERLAY

Use for sections with overlaying address ranges. If consecutive execution regions have the same `+<offset>`, then they are given the same base address.

The content is placed at a fixed address that does not change after linking. The content might overlap with other regions designated as `OVERLAY` regions.

This attribute modifies the behavior of the `--merge` and `--merge_litpools` command-line options.


Note

Arm Compiler for Embedded does not support using both manual and automatic overlays within the same program.

PADVALUE <value>

Defines the `<value>` to use for padding. If you specify `PADVALUE`, you must give a value, for example:

```
EXEC 0x10000 PADVALUE 0xFFFFFFFF EMPTY ZEROPAD 0x2000
```

This example creates a region of size `0x2000` that is filled with `0xFFFFFFFF`.

`PADVALUE` must be a word in size. `PADVALUE` attributes on load regions are ignored.

PI

This region contains only position independent sections. The content does not depend on any fixed address and might be moved after linking without any extra processing.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

This attribute is not supported if an image contains execute-only sections.

SORTTYPE <algorithm>

Specifies the sorting `<algorithm>` for the execution region, for example:

```
ER1 +0 SORTTYPE CallTree
```


Note

This attribute overrides any sorting algorithm that you specify with the `--sort` command-line option.

UNINIT

Use to create execution regions containing uninitialized data or memory-mapped I/O. Only ZI output sections are affected. For example, in the following ER_RW region only the ZI part is uninitialized:

```
LR 0x8000
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW 0x10000 UNINIT
    {
        *(+RW,+ZI)
    }
}
```


Note

Arm Compiler for Embedded does not support systems with ECC or parity protection where the memory is not initialized.

ZEROPAD

Zero-initialized sections are written in the ELF file as a block of zeros and, therefore, do not have to be zero-filled at runtime.

This attribute sets the load length of a ZI output section to `Image$$<region_name>$$ZI$$Length`.

Only root execution regions can be zero-initialized using the `ZEROPAD` attribute. Using the `ZEROPAD` attribute with a non-root execution region generates a warning and the attribute is ignored.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

In certain situations, such as a simulation, filling a region with a value is preferable to spending a long time in a zeroing loop.

Related information

Syntax of an execution region description on page 721
Behavior when .ANY sections overflow because of linker-generated content on page 681
Section alignment with the linker on page 594
Load$$ execution region symbols on page 625
Alignment of regions to page boundaries on page 693
Alignment of execution regions and input sections on page 694
Example of using expression evaluation in a scatter file to avoid padding on page 697
Example of aligning a base address in execution space but still tightly packed in load space on page 743
AlignExpr(expr, align) function on page 741
BNF notation used in scatter-loading description syntax on page 711
--any_contingency on page 425
Considerations when using a relative address +offset for execution regions on page 729
Expression evaluation in scatter files on page 735
Optimization with RW data compression on page 612
Image$$ execution region symbols on page 624
Syntax of an input section description on page 730
Inheritance rules for execution region address attributes on page 727
--merge, --no_merge on page 507
--merge_litpools, --no_merge_litpools on page 508
--overlay_veneers on page 510
--sort=algorithm on page 535
Overlay support in Arm Compiler for Embedded 6
Interaction of OVERLAY and PROTECTED attributes with armlink merge options

3.7.4.4 Inheritance rules for execution region address attributes

An execution region can inherit the attributes of a previous execution region.

For an execution region to inherit the attributes of a previous execution region, specify a `+<offset>` base address for that region. The first `+<offset>` execution region can inherit the attributes of the parent load region. An execution region cannot inherit attributes if:

- You explicitly set the attribute of that execution region.
- The previous execution region has the `AUTO_OVERLAY` or `OVERLAY` attribute.

You can explicitly set an execution region with the `ABSOLUTE`, `AUTO_OVERLAY`, `PI`, or `OVERLAY` attributes. However, an execution region can only inherit the `RELOC` attribute from the parent load region.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

PI and RELOC are not supported for AArch64 state.

The following inheritance rules apply when no address attribute is specified:

- The OVERLAY attribute cannot be inherited. A region with the OVERLAY attribute cannot inherit.
- A base address load or execution region always defaults to ABSOLUTE.
- A +<offset> execution region inherits the address attribute from the previous execution region or parent load region if no previous execution region exists.

Example

This example shows the inheritance rules for setting the address attributes of execution regions:

```
LR1 0x8000 PI
{
    ER1 +0          ; ER1 inherits PI from LR1
    {
        ...
    }
    ER2 +0          ; ER2 inherits PI from ER1
    {
        ...
    }
    ER3 0x10000     ; ER3 does not inherit because it has no relative base
                      address and gets the default of ABSOLUTE
    {
        ...
    }
    ER4 +0          ; ER4 inherits ABSOLUTE from ER3
    {
        ...
    }
    ER5 +0 PI       ; ER5 does not inherit, it explicitly sets PI
    {
        ...
    }
    ER6 +0 OVERLAY ; ER6 does not inherit, an OVERLAY cannot inherit
    {
        ...
    }
    ER7 +0         ; ER7 cannot inherit OVERLAY, gets the default of ABSOLUTE
    {
        ...
    }
}
```

Related information

Components of a load region description on page 713
Components of an execution region description on page 720
Considerations when using a relative address +offset for a load region on page 719
Inheritance rules for load region address attributes on page 717
Considerations when using a relative address +offset for execution regions on page 729
Syntax of an execution region description on page 721

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.7.4.5 Considerations when using a relative address +offset for execution regions

There are some considerations to be aware of when using a relative address for execution regions.

When using `+<offset>` to specify an execution region base address:

- The first execution region inherits the attributes of the parent load region, unless an attribute is explicitly set on that execution region.
- A `+<offset>` execution region ER2 inherits the attributes of the execution region ER1 immediately before it, unless:
  - ER1 has the `OVERLAY` attribute.
  - ER2 has an explicit attribute set.

  If an execution region is unable to inherit an attribute, then it gets the attribute `ABSOLUTE`.
- If the parent load region has the `RELOC` attribute, then all execution regions within that load region must have a `+<offset>` base address.

Related information

Inheritance rules for execution region address attributes on page 727
Inheritance rules for the RELOC address attribute on page 719

3.7.5 Input section descriptions

An input section description is a pattern that identifies input sections.

3.7.5.1 Components of an input section description

The components of an input section description allow you to identify the parts of an ELF file that are to be placed in an execution region.

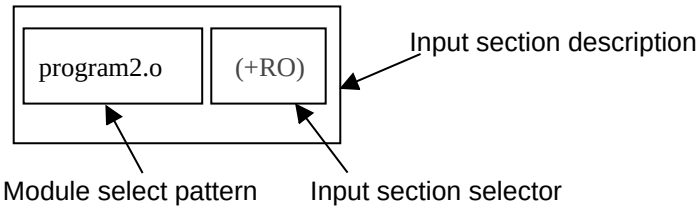
An input section description identifies input sections by:

- Module name (object filename, library member name, or library filename). The module name can use wildcard characters.
- Input section name, type, or attributes such as `READ-ONLY`, or `CODE`. You can use wildcard characters for the input section name.
- Symbol name.

The following figure shows the components of a typical input section description.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Figure 3-15: Components of an input section description



Note

Ordering in an execution region does not affect the ordering of sections in the output image.

Input section descriptions when linking partially-linked objects

You cannot specify partially-linked objects in an input section description, only the combined object file.

For example, if you link the partially linked objects `obj1.o`, `obj2.o`, and `obj3.o` together to produce `obj_all.o`, the component object names are discarded in the resulting object. Therefore, you cannot refer to one of the objects by name, for example, `obj1.o`. You can refer only to the combined object `obj_all.o`.

Related information

Syntax of an input section description on page 730
Syntax of a scatter file on page 711
--partial on page 515

3.7.5.2 Syntax of an input section description

An input section description specifies what input sections are loaded into the parent execution region.

The syntax of an input section description, in *Backus-Naur Form* (BNF), is:

```
<input_section_description> ::=
        <module_select_pattern> [ "(" <input_section_selector> ( ","
 <input_section_selector> )* ")" ]
<input_section_selector> ::= "+" <input_section_attr>
        | <input_section_pattern>
        | <input_section_type>
        | <input_symbol_pattern>
        | <section_properties>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Where:

<module_select_pattern>

A pattern that is constructed from literal text. An input section matches a module selector pattern when <module_select_pattern> matches one of the following:

- The name of the object file containing the section.
- The name of the library member, without a leading path name.
- The full name of the library, including the path name, from which the section is extracted. If the names contain spaces, use wild characters to simplify searching. For example, use *libname.lib to match C:\lib dir\libname.lib.

The wildcard character * matches zero or more characters and ? matches any single character.

Matching is not case-sensitive, even on hosts with case-sensitive file naming.

Use *.o to match all objects. Use * to match all object files and libraries.

You can use quoted filenames, for example "file one.o".

You cannot have two * selectors in a scatter file. You can, however, use two modified selectors, for example *A and *B, and you can use a .ANY selector together with a * module selector. The * module selector has higher precedence than .ANY. If the portion of the file containing the * selector is removed, the .ANY selector then becomes active.

<input_section_attr>

An attribute selector that is matched against the input section attributes. Each <input_section_attr> follows a +.

The selectors are not case-sensitive. The following selectors are recognized:

- RO-CODE.
- RO-DATA.
- RO, selects both RO-CODE and RO-DATA.
- RW-DATA.
- RW-CODE.
- RW, selects both RW-CODE and RW-DATA.
- XO.
- ZI.
- ENTRY, that is, a section containing an ENTRY point.

The following synonyms are recognized:

- CODE for RO-CODE.
- CONST for RO-DATA.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- TEXT for RO.
- DATA for RW.
- BSS for ZI.

The following pseudo-attributes are recognized:

- FIRST.
- LAST.

Use FIRST and LAST to mark the first and last sections in an execution region if the placement order is important. For example, if a specific input section must be first in the region and an input section containing a checksum must be last.



FIRST and LAST must not violate the basic attribute sorting order. For example, FIRST RW is placed after any read-only code or read-only data.

There can be only one FIRST or one LAST attribute for an execution region, and it must follow a single <input_section_selector>. For example:

***(section, +FIRST)**
This pattern is correct.

***(+FIRST, section)**
This pattern is incorrect and produces an error message.

<input_section_pattern>
A pattern that is matched, without case sensitivity, against the input section name. It is constructed from literal text. The wildcard character * matches 0 or more characters, and ? matches any single character. You can use a quoted input section name.



If you use more than one <input_section_pattern>, ensure that there are no duplicate patterns in different execution regions to avoid ambiguity errors.

<input_section_type>
A number that is compared against the input section type. The number can be decimal or hexadecimal.

<input_symbol_pattern>
You can select the input section by the global symbol name that the section defines. The global name enables you to choose individual sections with the same name from partially linked objects.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The `:gdef:` prefix distinguishes a global symbol pattern from a section pattern. For example, use `:gdef:mysym` to select the section that defines `mysym`. The following example shows a scatter file in which `ExecReg1` contains the section that defines global symbol `mysym1`, and the section that contains global symbol `mysym2`:

```
LoadRegion 0x8000
{
    ExecReg1 +0
    {
        *(:gdef:mysym1)
        *(:gdef:mysym2)
    }
    ; rest of scatter-loading description
}
```

You can use a quoted global symbol pattern. The `:gdef:` prefix can be inside or outside the quotes.


Note

If you use more than one `<input_symbol_pattern>`, ensure that there are no duplicate patterns in different execution regions to avoid ambiguity errors.

`<section_properties>`

A section property can be `+FIRST`, `+LAST`, and `OVERALIGN <value>`.

The value for `OVERALIGN` must be a positive power of 2 and must be greater than or equal to 4.

`armlink` does not `OVERALIGN` some sections where it might be unsafe to do so. In particular, sections that rely on or might rely on control falling through to adjacent sections, or that expect a table of contiguous sections to step through. For example, programs that generate a `PT_ARM_EXIDX` program header that describes the location of the contiguous range of `.arm.exidx` sections.

`armlink` does not `OVERALIGN`:

- A section with a linker defined `$$Base`, `$$Limit`, or `$$Length` symbol.
- A section with an inline veneer.
- A section with a link-order dependency on another section. That is, an ELF section header entry for a section that has the `SHF_LINK_ORDER` flag set. The `sh_link` field for such sections holds the index to another section header entry. Therefore, if a Section `s` has its `SHF_LINK_ORDER` flag set, and its `sh_link` field points to the index of Section `L`, then the linker must maintain this relative order between `s` and `L` in the output file.


Note

- The order of input section descriptors is not significant.
- Only input sections that match both `<module_select_pattern>` and at least one `<input_section_attr>` or `<input_section_pattern>` are included in the execution region.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If you omit `(+<input_section_attr>)` and `(<input_section_pattern>)`, the default is `+RO`.

- Do not rely on input section names that the compiler generates, or that are used by Arm library code. For example, if different compiler options are used, the input section names can change between compilations. In addition, section naming conventions that are used by the compiler are not guaranteed to remain constant between releases.
- The BNF definitions contain extra line returns and spaces to improve readability. If present in a scatter file, they are not required in scatter-loading descriptions and are ignored.

ELF section types and flags matched by each scatter-loading selector

The input section selectors for scatter-loading select based on the ELF section type (`SHT_*`) and flags (`SHF_*`). The following table shows the matching criteria:

Input section selectors	ELF section type	ELF section flag		
		SHF_EXECINSTR	SHF_WRITE	SHF_ARM_NOREAD
RO-CODE	SHT_PROGBITS	Y	-	-
RO-DATA	SHT_PROGBITS	-	-	-
RW-CODE	SHT_PROGBITS	Y	Y	-
RW-DATA	SHT_PROGBITS	-	Y	-
XO	SHT_PROGBITS	Y	-	Y
ZI	SHT_NOBITS	-	-	-

Related information

Relationship between the default armclang-generated sections and scatter-loading input sections on page 592
Components of an input section description on page 729
Behavior when .ANY sections overflow because of linker-generated content on page 681
Examples of module and input section specifications on page 734
BNF notation used in scatter-loading description syntax on page 711
Syntax of a scatter file on page 711
Examples of using placement algorithms for .ANY sections on page 675
Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 677
Examples of using sorting algorithms for .ANY sections on page 679
Alignment of execution regions and input sections on page 694
Manual placement of unassigned sections on page 670

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.7.5.3 Examples of module and input section specifications

Examples of `<module_select_pattern>` specifications and `<input_section_selector>` specifications.

Examples of `<module_select_pattern>` specifications are:

- `*` matches any module or library.
- `*.o` matches any object module.
- `math.o` matches the `math.o` module.
- `*armlib*` matches all C libraries supplied by Arm.
- `"file 1.o"` matches the file `file 1.o`.
- `*math.lib` matches any library path ending with `math.lib`, for example, `C:\apps\lib\math\satmath.lib`.

Examples of `<input_section_selector>` specifications are:

- `+RO` is an input section attribute that matches all RO code and all RO data.
- `+RW,+ZI` is an input section attribute that matches all RW code, all RW data, and all ZI data.
- `BLOCK_42` is an input section pattern that matches sections named `BLOCK_42`. There can be multiple ELF sections with the same `BLOCK_42` name that possess different attributes, for example `+RO-CODE,+RW`.

Related information

Components of an input section description on page 729
Syntax of an input section description on page 730

3.7.6 Expression evaluation in scatter files

Scatter files frequently contain numeric constants. These can be specific values, or the result of an expression.

3.7.6.1 Expression usage in scatter files

You can use expressions for various load and execution region attributes.

Expressions can be used in the following places:

- Load and execution region `<base_address>`.
- Load and execution region `+<offset>`.
- Load and execution region `<max_size>`.
- Parameter for the `ALIGN`, `FILL` or `PADVALUE` keywords.
- Parameter for the `ScatterAssert` function.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example of specifying the maximum size in terms of an expression

```
LR1 0x8000 (2 * 1024)
{
    ER1 +0 (1 * 1024)
    {
        *(+RO)
    }
    ER2 +0 (1 * 1024)
    {
        *(+RW,+ZI)
    }
}
```

Related information

Expression rules in scatter files on page 736
Execution address built-in functions for use in scatter files on page 737
ScatterAssert function and load address related functions on page 739
Symbol related function in a scatter file on page 740
Considerations when using a relative address +offset for a load region on page 719
Considerations when using a relative address +offset for execution regions on page 729
Example of aligning a base address in execution space but still tightly packed in load space on page 743
Syntax of a scatter file on page 711
Syntax of a load region description on page 714
Syntax of an execution region description on page 721

3.7.6.2 Expression rules in scatter files

Expressions follow the C-Precedence rules.

Expressions are made up of the following:

- Decimal or hexadecimal numbers.
- Arithmetic operators: `+`, `-`, `/`, `*`, `~`, `OR`, and `AND`

  The `OR` and `AND` operators map to the C operators `|` and `&` respectively.
- Logical operators: `LOR`, `LAND`, and `!`

  The `LOR` and `LAND` operators map to the C operators `||` and `&&` respectively.
- Relational operators: <, <=, >, >=, and ==

  Zero is returned when the expression evaluates to false and nonzero is returned when true.
- Conditional operator: `<Expression> ? <Expression1> : <Expression2>`

  This matches the C conditional operator. If `<Expression>` evaluates to nonzero then `<Expression1>` is evaluated otherwise `<Expression2>` is evaluated.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

When using a conditional operator in a `+<offset>` context on an execution region or load region description, the final expression is considered relative only if both `<Expression1>` and `<Expression2>`, are considered relative. For example:

```
er1 0x8000
{
    ...
}
er2 ((ImageLimit(er1) < 0x9000) ? +0 : +0x1000)    ; er2 has a
 relative address
{
    ...
}
er3 ((ImageLimit(er2) < 0x10000) ? 0x0 : +0)       ; er3 has an
 absolute address
{
    ...
}
```

- Functions that return numbers.

All operators match their C counterparts in meaning and precedence.

Expressions are not case-sensitive and you can use parentheses for clarity.

Related information

Expression usage in scatter files on page 735
Execution address built-in functions for use in scatter files on page 737
ScatterAssert function and load address related functions on page 739
Symbol related function in a scatter file on page 740
Considerations when using a relative address +offset for a load region on page 719
Considerations when using a relative address +offset for execution regions on page 729
Example of aligning a base address in execution space but still tightly packed in load space on page 743
Syntax of a scatter file on page 711
Syntax of a load region description on page 714
Syntax of an execution region description on page 721

3.7.6.3 Execution address built-in functions for use in scatter files

Built-in functions are provided for use in scatter files to calculate execution addresses.

The execution address related functions can only be used when specifying a `<base_address>`, `+<offset>` value, or `<max_size>`. They map to combinations of the linker defined symbols shown in the following table.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 3-23: Execution address related functions

Function	Linker defined symbol value
`ImageBase(<region_name>)`	`Image$$<region_name>$$Base`
`ImageLength(<region_name>)`	`Image$$<region_name>$$Length` + `Image$$<region_name>$$ZI$$Length`
`ImageLimit(<region_name>)`	`Image$$<region_name>$$Base` + `Image$$<region_name>$$Length` + `Image$$<region_name>$$ZI$$Length`

The parameter `<region_name>` can be either a load or an execution region name. Forward references are not permitted. The `<region_name>` can only refer to load or execution regions that have already been defined.

Note

You cannot use these functions when using the `.ANY` selector pattern. This is because a `.ANY` region uses the maximum size when assigning sections. The maximum size might not be available at that point, because the size of all regions is not known until after the `.ANY` assignment.

The following example shows how to use `ImageLimit(<region_name>)` to place one execution region immediately after another:

```
LR1 0x8000
{
    ER1 0x100000
    {
        *(+RO)
    }
}
LR2 0x100000
{
    ER2 (ImageLimit(ER1))               ; Place ER2 after ER1 has finished
    {
        *(+RW +ZI)
    }
}
```

Using `+offset` with expressions

A `+<offset>` value for an execution region is defined in terms of the previous region. You can use this as an input to other expressions such as `AlignExpr`. For example:

```
LR1 0x4000
{
    ER1 AlignExpr(+0, 0x8000)
    {
        ...
    }
}
```

By using `AlignExpr`, the result of `+0` is aligned to a `0x8000` boundary. This creates an execution region with a load address of `0x4000` but an execution address of `0x8000`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Expression usage in scatter files on page 735
Expression rules in scatter files on page 736
ScatterAssert function and load address related functions on page 739
Symbol related function in a scatter file on page 740
Considerations when using a relative address +offset for a load region on page 719
Scatter files containing relative base address load regions and a ZI execution region on page 744
Considerations when using a relative address +offset for execution regions on page 729
Syntax of a scatter file on page 711
Syntax of a load region description on page 714
Syntax of an execution region description on page 721
AlignExpr(expr, align) function on page 741
Image$$ execution region symbols on page 624
Example of aligning a base address in execution space but still tightly packed in load space on page 743

3.7.6.4 ScatterAssert function and load address related functions

The `ScatterAssert` function allows you to perform more complex size checks than those permitted by the `<max_size>` attribute.

The `ScatterAssert(<expression>)` function can be used at the top level, or within a load region. It is evaluated after the link has completed and gives an error message if `<expression>` evaluates to false.

The load address related functions can only be used within the `ScatterAssert` function. They map to the three linker defined symbol values:

Table 3-24: Load address related functions

Function	Linker defined symbol value
`LoadBase(<region_name>)`	`Load$$<region_name>$$Base`
`LoadLength(<region_name>)`	`Load$$<region_name>$$Length`
`LoadLimit(<region_name>)`	`Load$$<region_name>$$Limit`

The parameter `<region_name>` can be either a load or an execution region name. Forward references are not permitted. The `<region_name>` can only refer to load or execution regions that have already been defined.

The following example shows how to use the `ScatterAssert` function to write more complex size checks than those permitted by the `<max_size>` attribute of the region:

```
LR1 0x8000
{
    ER0 +0
    {
        *(+RO)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    }
    ER1 +0
    {
        file1.o(+RW)
    }
    ER2 +0
    {
        file2.o(+RW)
    }
    ScatterAssert((LoadLength(ER1) + LoadLength(ER2)) < 0x1000)
                                        ; LoadLength is compressed size
    ScatterAssert((ImageLength(ER1) + ImageLength(ER2)) < 0x2000)
                                        ; ImageLength is uncompressed size
}
ScatterAssert(ImageLength(LR1) < 0x3000)
                                        ; Check uncompressed size of load region
 LR1
```

Related information

Expression usage in scatter files on page 735
Expression rules in scatter files on page 736
Execution address built-in functions for use in scatter files on page 737
Symbol related function in a scatter file on page 740
Example of aligning a base address in execution space but still tightly packed in load space on page 743
Syntax of a scatter file on page 711
Syntax of a load region description on page 714
Syntax of an execution region description on page 721
Load$$ execution region symbols on page 625

3.7.6.5 Symbol related function in a scatter file

The symbol related function `defined` allows you to assign different values depending on whether or not a global symbol is defined.

The symbol related function, `defined(<global_symbol_name>)` returns zero if `<global_symbol_name>` is not defined and nonzero if it is defined.

Example: Conditionalizing a base address based on the presence of the symbol version1

The following scatter file shows an example of conditionalizing a base address based on the presence of the symbol `version1`:

```
LR1 0x8000
{
    ER1 (defined(version1) ? 0x8000 : 0x10000)   ; Base address is 0x8000
                                                 ; if version1 is defined
                                                 ; 0x10000 if not
    {
        *(+RO)
    }
    ER2 +0
    {
        *(+RW +ZI)
    }
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
}
```

Related information

Expression usage in scatter files on page 735
Expression rules in scatter files on page 736
Execution address built-in functions for use in scatter files on page 737
ScatterAssert function and load address related functions on page 739
Example of aligning a base address in execution space but still tightly packed in load space on page 743
Syntax of a scatter file on page 711
Syntax of a load region description on page 714
Syntax of an execution region description on page 721

3.7.6.6 AlignExpr(expr, align) function

Aligns an address expression to a specified boundary.

Syntax

```
AlignExpr(<expr>, <align>)
```

Parameters

<expr>

A valid address expression.

<align>

The alignment. Must be a power of 2.

Returns

This function returns:

```
(<expr> + (<align>-1)) & ~(<align>-1))
```

Operation

`AlignExpr` increases `<expr>` until:

```
<expr> ≡ 0 (mod <align>)
```

Relationship with the ALIGN keyword

The following relationship exists between `ALIGN` and `AlignExpr`:

ALIGN keyword

Load and execution regions already have an `ALIGN` keyword:

- For load regions the `ALIGN` keyword aligns the base of the load region in load space and in the file to the specified alignment.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- For execution regions the `ALIGN` keyword aligns the base of the execution region in execution and load space to the specified alignment.

`AlignExpr`

Aligns the expression it operates on, but has no effect on the properties of the load or execution region.

Example: Aligning on an 8-byte boundary

This example aligns the address of `ER2` on an 8-byte boundary:

```
ER +0
{
    ...
}
ER2 AlignExpr(+0x8000, 8)
{
    ...
}
```

Related information

Load region attributes on page 715
Execution region attributes on page 723
Linker support for creating demand-paged files on page 595

3.7.6.7 GetPageSize() function

Returns the page size when an image is demand-paged, and is useful when used with the `AlignExpr` function.

Default

By default the internal page size is set to `0x8000`, but you can change it with the `--pagesize` command-line option.

Syntax

```
GetPageSize()
```

Parameters

None.

Returns

When you link with the `--paged` command-line option, returns the value of the internal page size that `armlink` uses in its alignment calculations. Otherwise, it returns zero.

Example: Aligning to a page boundary

This example aligns the base address of `ER` to a page boundary:

```
ER AlignExpr(+0, GetPageSize())
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
{
    ...
}
```

Related information

Example of aligning a base address in execution space but still tightly packed in load space on page 743
--pagesize=pagesize on page 514
Linker support for creating demand-paged files on page 595
AlignExpr(expr, align) function on page 741

3.7.6.8 SizeOfHeaders() function

This function is useful when writing demand-paged images to start code and data immediately after the ELF header and Program Header table.

Syntax

```
SizeOfHeaders()
```

Parameters

None.

Returns

Returns the size of ELF header plus the estimated size of the Program Header table.

Example

This example sets the base of `LR1` to start immediately after the ELF header and Program Headers:

```
LR1 SizeOfHeaders()
{
    ...
}
```

Related information

Example of aligning a base address in execution space but still tightly packed in load space on page 743
Linker support for creating demand-paged files on page 595
Alignment of regions to page boundaries on page 693

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.7.6.9 Example of aligning a base address in execution space but still tightly packed in load space

This example shows how to use a combination of preprocessor macros and expressions to copy tightly packed execution regions to execution addresses in a page-boundary.

Using the `ALIGN` scatter-loading keyword aligns the load addresses of `ER2` and `ER3` as well as the execution addresses

Aligning a base address in execution space but still tightly packed in load space

```
#! armclang -E
#define START_ADDRESS  0x100000
#define PAGE_ALIGNMENT 0x100000

LR1 0x8000
{
    ER0 +0
    {
        *(InRoot$$Sections)
    }
    ER1 START_ADDRESS
    {
        file1.o(*)
    }
    ER2 AlignExpr(ImageLimit(ER1), PAGE_ALIGNMENT)
    {
        file2.o(*)
    }
    ER3 AlignExpr(ImageLimit(ER2), PAGE_ALIGNMENT)
    {
        file3.o(*)
    }
}
```

Related information

Load region attributes on page 715
Execution region attributes on page 723
GetPageSize() function on page 742
SizeOfHeaders() function on page 743
Syntax of a load region description on page 714
Syntax of an execution region description on page 721
AlignExpr(expr, align) function on page 741

3.7.6.10 Scatter files containing relative base address load regions and a ZI execution region

You might want to place *Zero-Initialized* (ZI) data in one load region, and use a relative base address for the next load region.

To place ZI data in load region LR1, and use a relative base address for the next load region LR2, for example:

```
LR1 0x8000
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
{
    er_progbits +0
    {
        *(+RO,+RW) ; Takes space in the Load Region
    }
    er_zi +0
    {
        *(+ZI) ; Takes no space in the Load Region
    }
}
LR2 +0 ; Load Region follows immediately from LR1
{
    er_moreprogbits +0
    {
        file1.o(+RO) ; Takes space in the Load Region
    }
}
```

Because the linker does not adjust the base address of LR2 to account for ZI data, the execution region `er_zi` overlaps the execution region `er_moreprogbits`. This generates an error when linking.

To correct this, use the `ImageLimit()` function with the name of the ZI execution region to calculate the base address of LR2. For example:

```
LR1 0x8000
{
    er_progbits +0
    {
        *(+RO,+RW) ; Takes space in the Load Region
    }
    er_zi +0
    {
        *(+ZI) ; Takes no space in the Load Region
    }
}
LR2 ImageLimit(er_zi) ; Set the address of LR2 to limit of er_zi
{
    er_moreprogbits +0
    {
        file1.o(+RO) ; Takes space in the Load Region
    }
}
```

Related information

Expression evaluation in scatter files on page 735
Syntax of a scatter file on page 711
Syntax of a load region description on page 714
Syntax of an execution region description on page 721
Expression usage in scatter files on page 735
Expression rules in scatter files on page 736
Image$$ execution region symbols on page 624
Execution address built-in functions for use in scatter files on page 737

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.8 SysV Shared Libraries and Executables

Describes how the Arm linker, `armlink`, supports the *System V* (SysV) shared libraries and executables.

3.8.1 SysV linking model

System V (SysV) files have a standard linking model that is described in the generic ELF specification.

There are several platform operating systems that use the SysV format, for example, Arm® Linux.

3.8.1.1 SysV standard memory model

When you use the `--sysv` command-line option, the linker automatically applies the SysV standard memory model.

This is equivalent to the following image layout:

```
LR_1 <read-only base address> + SizeOfHeaders()
{
    .interp +0
    {
        *(.interp)
    }
    .note.ABI-tag +0
    {
        *(.note.ABI-tag)
    }
    .hash +0
    {
        *(0x00000005) ; SHT_HASH
    }
    .dynsym +0
    {
        *(0x0000000b) ; SHT_DYNSYM
    }
    .dynstr +0
    {
        *(0x00000003) ; SHT_STRTAB
    }
    .version +0
    {
        *(0x6fffffff) ; SHT_GNU_versym
    }
    .version_d +0
    {
        *(0x6ffffffd) ; SHT_GNU_verdef
    }
    .version_r +0
    {
        *(0x6ffffffe) ; SHT_GNU_verneed
    }
    .rel.dyn +0
    {
        *(.rel.dyn)
    }
    .rela.dyn +0
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    {
        *(.rela.dyn)
    }
    .rel.plt +0
    {
        *(.rel.plt)
    }
    .rela.plt +0
    {
        *(.rela.plt)
    }
    .init +0
    {
        *(.init)
    }
    .plt +0
    {
        *(.plt)
    }
    .text +0
    {
        *(+RO)
    }
    .fini +0
    {
        *(.fini)
    }
    .ARM.exidx +0
    {
        *(0x70000001) ; SHT_ARM_EXIDX
    }
    .eh_frame_hdr +0
    {
        *(.eh_frame_hdr)
    }
}
LR_2 ImageLimit(LR_1) == AlignExpr(ImageLimit(LR_1), GetPageSize())
? +0
: +GetPageSize()
{
    .tdata +0
    {
        *(+TLS-RW)
    }
    .tbss +0
    {
        *(+TLS-ZI)
    }
    .preinit_array +0
    {
        *(0x00000010) ; SHT_PREINIT_ARRAY
    }
    .init_array +0
    {
        *(0x0000000e) ; SHT_INIT_ARRAY
    }
    .fini_array +0
    {
        *(0x0000000f) ; SHT_FINI_ARRAY
    }
    .dynamic +0
    {
        *(0x00000006) ; SHT_DYNAMIC
    }
    .got +0
    {
        *(.got)
    }
    .data +0
    {
        *(+RW)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    }
    .bss +0
    {
        *(+ZI)
    }
}
```

The `<read-only base address>` is controlled by the `--ro_base` command-line option. You can use the `--scatter=<filename>` option with SysV to specify a custom memory layout.

Related information

Execution address built-in functions for use in scatter files on page 737
AlignExpr(expr, align) function on page 741
GetPageSize() function on page 742
SizeOfHeaders() function on page 743

3.8.1.2 Requirements and restrictions for using scatter files with SysV linking model

There are requirements and restrictions for using scatter files with the SysV linking model, `--sysv`.

Requirements

In the paged-mapped model:

- The ELF header and program headers are included in the first load region. `armlink` cannot automatically determine the space needed for the ELF header and program headers. Therefore, the scatter file must use the `SizeOfHeaders()` function to explicitly instruct the linker to leave this space.
- All load region base addresses must be aligned to a page-sized boundary.

See SysV standard memory model for the scatter file layout that `armlink` uses to apply the SysV standard memory model.

Restrictions

The following restrictions apply to scatter files for the SysV linking model:

- All non-`ZI` execution regions must be root regions.
- You must place any `ZI` execution regions at the end of a load region.
- You cannot use the `OVERLAY`, `PI`, or `RELOC` execution region attributes.
- If you are linking without `--bare_metal_sysv` for a bare-metal application, you must not use a scatter file to initialize the stack and the heap. Therefore, you cannot use any of the following execution region names:
  - `ARM_LIB_HEAP`
  - `ARM_LIB_STACK`
  - `ARM_LIB_STACKHEAP`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--bare_metal_sysv on page 430
--scatter=filename on page 529
Execution region attributes on page 723
Root regions

3.8.1.3 Using the C and C++ libraries

You can use either the Arm C and C++ libraries or platform libraries with the SysV linking model.

Use of the Arm C and C++ libraries

You can use the Arm C and C++ libraries with the SysV linking model by statically linking the main executable with them. You must appropriately retarget the library for the platform.


Note

When performing the standard library selection as described in How the linker searches for the Arm standard libraries, the linker selects the best-suited variants of the C and C++ libraries with the SysV linking model by statically linking the main executable with them. You must appropriately retarget the library for the platform. Arm C and C++ libraries based only on the attributes of input objects that are used to build the main executable. Shared libraries used in the link and their input objects do not affect the library selection.

Integration with a dynamic loader

- The Arm C and C++ libraries with the SysV linking model by statically linking the main C library executes pre-initialization (`.preinit_array`) and initialization functions (`.init_array`) that are present only in the main executable. The library is not aware of initialization functions in loaded shared objects.

  To enable running initialization routines in the whole program, you can link the main executable with `armlink --no_preinit --no_cppinit` and provide custom implementation of `__arm_preinit_()` and `__cpp_initialize__aeabi_()`. The overridden functions must integrate with a platform dynamic loader to execute all initialization functions.

  The dynamic loader can use dynamic entries `DT_PREINIT_ARRAY`, `DT_INIT_ARRAY`, `DT_INIT` to obtain initialization functions in the executable and each shared object.

- The Arm C++ library by default supports exceptions only in the main executable. To allow exceptions in loaded shared objects, you can provide implementation of `__arm_find_exidx_section()` (in AArch32 state) and `__arm_find_eh_frame_hdr_section()` (in AArch64 state):

```
/* AArch32 hook */
int __arm_find_exidx_section(uintptr_t target_addr, uintptr_t *base, size_t
 *length);

/* AArch64 hook */
int __arm_find_eh_frame_hdr_section(uintptr_t target_addr, uintptr_t *base,
 size_t *length);
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The functions receive an address of code that needs to be unwound and must find an exception-index section associated with this location. Parameter `target_addr` specifies an address of code that needs to be unwound. Parameters `base` and `length` point to values that must be set by the function to the address and size of the found exception-index section. Return value `0` indicates success, non-zero value indicates a failure. The dynamic loader can use segments `PT_ARM_EXIDX` (in AArch32 state) and `PT_GNU_EH_FRAME` (in AArch64 state) to locate the exception-index sections.

Use of the platform C and C++ libraries

It is possible to use system libraries that come with the target platform.

To indicate to the compiler to not use the Arm C and C++ libraries, compile the code of the program with the `armclang` command-line options `-nostdlib` and `-nostdlibinc`.

You must link executable and shared objects with the `armlink` command-line option `--no_scanlib`.

Related information

-nostdlib on page 200
-nostdlibinc on page 201

3.8.1.4 Using a dynamic Linker

A shared object or executable file contains all the information necessary for a dynamic linker to load and run the file correctly.

- Every shared object contains a `SONAME` that identifies the object. You can specify this name by using the `--soname=<name>` command-line option.
- The linker identifies dependencies to other shared objects using the shared objects specified on the command line. These shared object dependencies are encoded in `DT_NEEDED` tags. The linker orders these tags to match the order of the libraries on the command line.
- If you specify the `--init` symbol command-line option, the linker uses the specified symbol name to define initialization code and records its address in the `DT_INIT` tag. The dynamic linker must execute this code when it loads the executable file or shared object.
- If you specify the `--fini` symbol command-line option, the linker uses the specified symbol name to define termination code and records its address in the `DT_FINI` tag. The dynamic linker executes this code when it unloads the executable file or shared object.

Use the `--dynamiclinker=<name>` command-line option to specify the dynamic linker to use to load and relocate the file at runtime.

3.8.1.5 Automatic dynamic symbol table rules in the SysV linking model

There are rules that apply to dynamic symbol tables for the *System V* (SysV) linking model.

The following rules apply:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Executable

An undefined symbol reference is an undefined symbol error.

Global symbols with `STV_HIDDEN` or `STV_INTERNAL` visibility are never exported to the dynamic symbol table.

Global symbols with `STV_PROTECTED` or `STV_DEFAULT` visibility are not exported to the dynamic symbol table unless you specify the `--export_all` or `--export_dynamic` option.

Shared library

An undefined symbol reference with `STV_DEFAULT` visibility is treated as imported and is placed in the dynamic symbol table.

An undefined symbol reference without `STV_DEFAULT` visibility is an undefined symbol error.

Global symbols with `STV_HIDDEN` or `STV_INTERNAL` visibility are never exported to the dynamic symbol table.


Note

`STV_HIDDEN` or `STV_INTERNAL` global symbols that are required for relocation can be placed in the dynamic symbol table, however the linker changes them into local symbols to prevent them from being accessed from outside the shared library.

Global symbols with `STV_PROTECTED` or `STV_DEFAULT` visibility are always exported to the dynamic symbol table.

Related information

Addressing modes in the SysV linking model on page 752

3.8.1.6 Symbol definitions defined for SysV compatibility with glibc

To improve *System V* (SysV) compatibility with `glibc`, the linker defines various symbols.

The linker defines the following symbols if the corresponding sections exist in an object:

- For `.init_array` sections:
  - `__init_array_start`.
  - `__init_array_end`.
- For `.fini_array` sections:
  - `__fini_array_start`.
  - `__fini_array_end`.
- For `.ARM.exidx` sections:
  - `__exidx_start`.
  - `__exidx_end`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- For `.preinit_array` sections:
  - `__preinit_array_start`.
  - `__preinit_array_end`.
- `__executable_start`.
- `etext`.
- `_etext`.
- `__etext`.
- `__data_start`.
- `edata`.
- `_edata`.
- `__bss_start`.
- `__bss_start__`.
- `_bss_end__`.
- `__bss_end__`.
- `end`.
- `_end`.
- `__end`.
- `__end__`.

Related information

SysV linking model on page 746
ELF for the Arm Architecture

3.8.1.7 Addressing modes in the SysV linking model

System V (SysV) has a defined model for accessing the program and imported data and code from other modules.

If required, the linker automatically generates the required *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT) sections.

Position independent code

SysV shared libraries are compiled with *Position Independent Code* (PIC) using the `-fpic` compiler command-line option.

You must also use the linker command-line option `--fpic` to declare that a shared library is position independent because this affects the construction of the PLT and GOT sections.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

By default, the linker produces an error message if the command-line option `--shared` is given without the `--fpic` options. If you must create a shared library that is not position independent, you can turn the error message off by using `--diag_suppress=6403`.

Related information

Automatic dynamic symbol table rules in the SysV linking model on page 750

3.8.1.8 Thread local storage in the SysV linking model

Thread Local Storage (TLS) support in `armlink` depends on whether you are building for AArch32 or AArch64.

AArch32

`armlink` supports the traditional Arm Linux TLS model in AArch32 state. The *Addenda to*, and *Errata in*, the ABI for the Arm® Architecture describes the Arm Linux TLS model.

Note

`armlink` does not support the newer TLS descriptor model when building for AArch32. The *Application Binary Interface* (ABI) *ELF for the Arm Architecture* describes the *New experimental TLS relocations* used by this model.

AArch64

`armlink` supports the descriptor model of TLS when building for AArch64 state. The ELF for the 64-bit Arm Architecture describes the TLS relocations.

TLS relaxations are a mechanism to convert more general dynamic accesses into simpler, more restrictive, accesses. This is an optimization performed by the linker. The main optimization lies in removing the need for a *Global Offset Table* (GOT) slot and for any dynamic relocation to be processed at program load time.

In certain circumstances `armlink` can relax a code sequence that uses a general purpose, less restrictive, TLS model to a more efficient, but more restrictive, TLS model. Relaxation is most effective when the code models (`-mcmodel`) and TLS size (`-mtls-size`) have their default values.

`armlink` supports the following relaxations:

- `global-dynamic` to `local-exec`. `global-dynamic` is also known as the general dynamic model.
- `initial-exec` to `local-exec`.

Note

Relaxations are only supported with `-mcmodel=small`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

`armlink` does not support the `initial-exec` model when dynamic linking.

Related information

SysV linking model on page 746
-ftls-model on page 105
-mcmodel on page 146
-mtls-size on page 194
Addenda to, and Errata in, the ABI for the Arm Architecture (ABI-addenda)

3.8.1.9 Thread local storage in the bare metal and shared library linking models

Thread local variables are placed in ELF sections with the `SHF_TLS` flag. For *Thread Local Storage* (TLS) to work the linker must collate these variables into a single contiguous block called the TLS template.

The TLS template is instantiated once per thread so that each thread gets its own instance of each thread local variable. The TLS template must be in the form of two execution regions that are contiguous in memory. The first execution region contains the read-write TLS and the second contains the zero-initialized TLS.

The following scatter file fragment produces a TLS template in the required form:

```
ER_TLS_RW <address>
{
  *(+TLS-RW)
}
ER_TLS_ZI +0
{
  *(+TLS-ZI)
}
```

The SysV standard memory model has an example with the TLS template residing in `LR_2`.

Instantiating the TLS per thread is the responsibility of the program. When linking with `--sysv` the linker outputs a program header of type `PT_TLS` that can be used to locate the TLS template. When linking a bare-metal application, linker defined symbols can be used to locate the TLS template. You can use the following expressions to initialize the TLS template:

Table 3-25: Expressions for initializing the TLS template

TLS template component	PT_TLS field	Expression using linker-defined symbols
Start of TLS template	`p_vaddr`	`Image$$ER_TLS_RW$$Base`
Size of TLS RW	`p_filesz`	`Image$$ER_TLS_RW$$Limit` - `Image$$ER_TLS_RW$$Base`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

TLS template component	PT_TLS field	Expression using linker-defined symbols
Size of TLS ZI	`p_memsz - p_filesz`	`Image$$ER_TLS_ZI$$ZI$$Limit` - `Image$$ER_TLS_RW$$Limit`
Size of TLS template	`p_memsz`	`Image$$ER_TLS_ZI$$ZI$$Limit` - `Image$$ER_TLS_RW$$Base`

Note

Do not use `Image$$ER_TLS_ZI$$Length` for the size of TLS ZI because this does not include any additional padding added to align `ER_TLS_ZI`.

Note

It is the responsibility of the program or program loader to instantiate the TLS template for each thread.

Related information

-ftls-model on page 105
-mcmodel on page 146
-mtls-size on page 194
Addenda to, and Errata in, the ABI for the Arm Architecture (ABI-addenda)

3.8.1.10 Linker command-line options for the SysV linking model

There are linker command-line options available for the SysV linking model.

The linker command-line options are:

- `--bare_metal_sysv`
- `--dynamic_linker`.
- `--export_all`, `--no_export_all`.
- `--export_dynamic`, `--no_export_dynamic`.
- `--force_so_throw`, `--no_force_so_throw`.
- `--fpic`.
- `--import_unresolved`, `--no_import_unresolved`.
- `--pagesize=<pagesize>`.
- `--soname=<name>`.
- `--shared`.
- `--sysv`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

armlink Command-line Options on page 425

3.8.2 Symbol versioning

Symbol versioning records extra information about symbols imported from, and exported by, a dynamic shared object.

A dynamic loader uses this extra information to ensure that all the symbols required by an image are available at load time.

3.8.2.1 Overview of symbol versioning

Symbol versioning enables shared object creators to produce new versions of symbols for use by all new clients, while maintaining compatibility with clients linked against old versions of the shared object.

Version

Symbol versioning adds the concept of a version to the dynamic symbol table. A version is a name that symbols are associated with. When a dynamic loader tries to resolve a symbol reference associated with a version name, it can only match against a symbol definition with the same version name.



Note

A version might be associated with previous version names to show the revision history of the shared object.

Default version

While a shared object might have multiple versions of the same symbol, a client of the shared object can only bind against the latest version.

This is called the default version of the symbol.

Creation of versioned symbols

By default, the linker does not create versioned symbols for a non *Base Platform Application Binary Interface* (BPABI) shared object.

Related information

The symbol versioning script file on page 757

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.8.2.2 Embedded symbols

You can add specially-named symbols to input objects that cause the linker to create symbol versions.

These symbols are of the form:

- `name@version` for a non-default version of a symbol.
- `name@@version` for a default version of a symbol.

You must define these symbols, at the address of the function or data, as that you want to export. The symbol name is divided into two parts, a symbol name `<name>` and a version definition `<version>`. The `<name>` is added to the dynamic symbol table and becomes part of the interface to the shared object. Version creates a version called `<ver>` if it does not already exist and associates `<name>` with the version called `<ver>`.

Example: Place symbols into the object symbol table

The example in Example of creating versioned symbols shows how to place symbols into the object symbol table.

3.8.2.3 The symbol versioning script file

You can embed the commands to produce symbol versions in a script file.

You specify a symbol versioning script file with the command-line option `--symver_script=<file>`. Using this option automatically enables symbol versioning.

The script file supports the same syntax as the GNU *ld* linker.

Using a script file enables you to associate a version with an earlier version.

You can provide a steering file in addition to the embedded symbol method. If you choose to do this then your script file must match your embedded symbols and use the Backus-Naur Form (BNF) notation:

```
<version_definition> ::=
  <version_name> "{" <symbol_association>* "}" [<depend_version>] ";"

<symbol_association> ::=
  "local:" | "global:" | <symbol_name> ";"
```

Where:

- `<version_name>` is a string containing the name of the version.
- `<depend_version>` is a string containing the name of a version that this `<version_name>` depends on. This version must have already been defined in the script file.
- `"local:"` indicates that all subsequent `<symbol_name>`s in this version definition are local to the shared object and are not versioned.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- `"global:"` indicates that all subsequent `<symbol_name>`s belong to this version definition.

  There is an implicit `"global:"` at the start of every version definition.
- `<symbol_name>` is the name of a global symbol in the static symbol table.

Version names have no specific meaning, but they are significant in that they make it into the output. In the output, they are a part of the version specification of the library and a part of the version requirements of a program that links against such a library. The following example shows the use of version names:

```
VERSION_1
{
    ...
};
VERSION_2
{
    ...
} VERSION_1;
```


Note

If you use a script file then the version definitions and symbols associated with them must match. The linker warns you if it detects any mismatch.

Related information

Overview of symbol versioning on page 756

3.8.2.4 Example of creating versioned symbols

This example shows how to create versioned symbols in code and with a script file.

The following example places the symbols `testA@ver1`, `testA@@ver2`, and `testB@@ver1` into the object symbol table:

```
int old_function(void) __asm__("testA@ver1");
int new_function(void) __asm__("testA@@ver2");
int other_function(void) __asm__("testB@@ver1");
```

The linker reads these symbols and creates version definitions `ver1` and `ver2`. The symbol `testA` is associated with a non-default version of `ver1`, and with a default version of `ver2`. The symbol `testB` is associated with a default version of `ver1`.

There is no way to create associations between versions with this method. However, you can create dependencies using a symbol versioning script file.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The corresponding symbol versioning script file that includes the addition of dependency information so that `ver2` depends on `ver1` is:

```
ver1
{
    global:
        testA; testB;
    local:
        *;
};
ver2
{
    global:
        testA;
} ver1;
```

Related information

Embedded symbols on page 756
Symbol versioning on page 756
Linker options for enabling implicit symbol versioning on page 759
--symver_script=filename on page 547
Writing A32/T32 Instructions in armasm Syntax Assembly Language on page 904

3.8.2.5 Linker options for enabling implicit symbol versioning

If you have to version your symbols to force static binding, but you do not care about the version number that they are given, you can use implicit symbol versioning.

Use the command-line option `--symver_soname` to turn on implicit symbol versioning.

Where a symbol has no defined version, the linker uses the `SONAME` of the file being linked.

This option can be combined with embedded symbols or a script file. `armlink` adds the `SONAME { *; };` definition to its internal representation of a symbol versioning script.

Related information

The symbol versioning script file on page 757

3.9 Linker Steering File Command Reference

Describes the steering file commands supported by the Arm linker, `armlink`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.9.1 EXPORT steering file command

Specifies that a symbol can be accessed by other shared objects or executables.


Note

A symbol can be exported only if the definition has `STV_DEFAULT` or `STV_PROTECTED` visibility. You must use the `--override_visibility` command-line option to enable the linker to override symbol visibility to `STV_DEFAULT`.

Syntax

```
EXPORT <pattern> AS <replacement_pattern>[,<pattern> AS <replacement_pattern>]
```

where:

`<pattern>`

is a string, optionally including wildcard characters (either `*` or `?`), that matches zero or more defined global symbols. If `<pattern>` does not match any defined global symbol, the linker ignores the command. The operand can match only defined global symbols.

If the symbol is not defined, the linker issues:

```
Warning: L6331W: No eligible global symbol matches pattern symbol
```

`<replacement_pattern>`

is a string, optionally including wildcard characters (either `*` or `?`), to which the defined global symbol is to be renamed. Wild characters must have a corresponding wildcard in `<pattern>`. The characters matched by the `<replacement_pattern>` wildcard are substituted for the `<pattern>` wildcard.

For example:

```
EXPORT my_func AS func1
```

renames and exports the defined symbol `my_func` as `func1`.

Usage

You cannot export a symbol to a name that already exists. Only one wildcard character (either `*` or `?`) is permitted in `EXPORT`.

The defined global symbol is included in the dynamic symbol table (as `<replacement_pattern>` if given, otherwise as `<pattern>`), if a dynamic symbol table is present.

Related information

IMPORT steering file command on page 761
Edit the symbol tables with a steering file on page 639
--override_visibility on page 511

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.9.2 HIDE steering file command

Makes defined global symbols in the symbol table anonymous.

Syntax

```
HIDE <pattern>[,<pattern>]
```

where:

<pattern>

is a string, optionally including wildcard characters, that matches zero or more defined global symbols. If `<pattern>` does not match any defined global symbol, the linker ignores the command. You cannot hide undefined symbols.

Usage

You can use `HIDE` and `SHOW` to make certain global symbols anonymous in an output image or partially linked object. Hiding symbols in an object file or library can be useful as a means of protecting intellectual property, as shown in the following example:

```
; steer.txt
; Hides all global symbols
HIDE *
; Shows all symbols beginning with 'os_'
SHOW os_*
```

This example produces a partially linked object with all global symbols hidden, except those beginning with `os_`.

Link this example with the command:

```
armlink --partial input_object.o --edit steer.txt -o partial_object.o
```

You can link the resulting partial object with other objects, provided they do not contain references to the hidden symbols. When symbols are hidden in the output object, `SHOW` commands in subsequent link steps have no effect on them. The hidden references are removed from the output symbol table.

Related information

SHOW steering file command on page 766
--edit=file_list on page 455
--partial on page 515
Edit the symbol tables with a steering file on page 639

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.9.3 IMPORT steering file command

Specifies that a symbol is defined in a shared object at runtime.


Note

A symbol can be imported only if the reference has `STV_DEFAULT` visibility. You must use the `--override_visibility` command-line option to enable the linker to override symbol visibility to `STV_DEFAULT`.

Syntax

```
IMPORT <pattern> AS <replacement_pattern>[,<pattern> AS <replacement_pattern>]
```

where:

`<pattern>`

is a string, optionally including wildcard characters (either `*` or `?`), that matches zero or more undefined global symbols. If `<pattern>` does not match any undefined global symbol, the linker ignores the command. The operand can match only undefined global symbols.

`<replacement_pattern>`

is a string, optionally including wildcard characters (either `*` or `?`), to which the symbol is to be renamed. Wild characters must have a corresponding wildcard in `<pattern>`. The characters matched by the `<pattern>` wildcard are substituted for the `<replacement_pattern>` wildcard.

For example:

```
IMPORT my_func AS func
```

imports and renames the undefined symbol `my_func` as `func`.

Usage

You cannot import a symbol that has been defined in the current shared object or executable. Only one wildcard character (either `*` or `?`) is permitted in `IMPORT`.

The undefined symbol is included in the dynamic symbol table (as `<replacement_pattern>` if given, otherwise as `<pattern>`), if a dynamic symbol table is present.


Note

The `IMPORT` command only affects undefined global symbols. Symbols that have been resolved by a shared library are implicitly imported into the dynamic symbol table. The linker ignores any `IMPORT` directive that targets an implicitly imported symbol.

Related information

--override_visibility on page 511
EXPORT steering file command on page 759
Edit the symbol tables with a steering file on page 639

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.9.4 RENAME steering file command

Renames defined and undefined global symbol names.

Syntax

```
RENAME <pattern> AS <replacement_pattern>[,<pattern> AS <replacement_pattern>]
```

where:

<pattern>

is a string, optionally including wildcard characters (either * or ?), that matches zero or more global symbols. If <pattern> does not match any global symbol, the linker ignores the command. The operand can match both defined and undefined symbols.

<replacement_pattern>

is a string, optionally including wildcard characters (either * or ?), to which the symbol is to be renamed. Wildcard characters must have a corresponding wildcard in <pattern>. The characters matched by the <pattern> wildcard are substituted for the <replacement_pattern> wildcard.

For example, for a symbol named func1:

```
RENAME f* AS my_f*
```

renames func1 to my_func1.

Usage

You cannot rename a symbol to a global symbol name that already exists, even if the target symbol name is being renamed itself.

You cannot rename a symbol to the same name as another symbol. For example, you cannot do the following:

```
RENAME foo1 AS bar
RENAME foo2 AS bar

Error: L6281E: Cannot rename both foo2 and foo1 to bar.
```

Renames only take effect at the end of the link step. Therefore, renaming a symbol does not remove its original name. For example, given an image containing the symbols func1 and func2, you cannot do the following:

```
RENAME func1 AS func2
RENAME func2 AS func3

Error: L6282E: Cannot rename func1 to func2 as a global symbol of that name exists
```

Only one wildcard character (either * or ?) is permitted in RENAME.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

Given an image containing the symbols `func1`, `func2`, and `func3`, you might have a steering file containing the following commands:

```
; invalid, func2 already exists
RENAME func1 AS func2

; valid
RENAME func3 AS b2

; invalid, func3 still exists because the link step is not yet complete
RENAME func2 AS func3
```

Related information

Edit the symbol tables with a steering file on page 639

3.9.5 REQUIRE steering file command

Creates a `DT_NEEDED` tag in the dynamic array.

`DT_NEEDED` tags specify dependencies to other shared objects used by the application, for example, a shared library.

Syntax

```
REQUIRE <pattern>[,<pattern>]
```

where:

`<pattern>`

is a string representing a filename. No wild characters are permitted.

Usage

The linker inserts a `DT_NEEDED` tag with the value of `<pattern>` into the dynamic array. This tells the dynamic loader that the file it is currently loading requires `<pattern>` to be loaded.



Note

`DT_NEEDED` tags inserted as a result of a `REQUIRE` command are added after `DT_NEEDED` tags generated from shared objects or dynamically linked libraries (DLLs) placed on the command line.

Related information

Edit the symbol tables with a steering file on page 639

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

3.9.6 RESOLVE steering file command

Matches specific undefined references to a defined global symbol.

Syntax

```
RESOLVE <pattern> AS <defined_pattern>
```

where:

<pattern>

is a string, optionally including wildcard characters (either `*` or `?`), that matches zero or more undefined global symbols. If `<pattern>` does not match any undefined global symbol, the linker ignores the command. The operand can match only undefined global symbols.

<defined_pattern>

is a string, optionally including wildcard characters, that matches zero or more defined global symbols. If `<defined_pattern>` does not match any defined global symbol, the linker ignores the command. You cannot match an undefined reference to an undefined symbol.

Usage

`RESOLVE` is an extension of the existing `armlink --unresolved` command-line option. The difference is that `--unresolved` enables all undefined references to match one single definition, whereas `RESOLVE` enables more specific matching of references to symbols.

The undefined references are removed from the output symbol table.

`RESOLVE` works when performing partial-linking and when linking normally.

Example

You might have two files `file1.c` and `file2.c`, as shown in the following example:

```
file1.c
extern int foo;
extern void MP3_Init(void);
extern void MP3_Play(void);
int main(void)
{
  int x = foo + 1;
  MP3_Init();
  MP3_Play();
  return x;
}

file2.c:
int bar;
void MyMP3_Init()
{
}
void MyMP3_Play()
{
}
```

Create a steering file, `ed.txt`, containing the line:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
RESOLVE MP3* AS MyMP3*
```

Enter the following command:

```
armlink file1.o file2.o --edit ed.txt --unresolved bar
```

This command has the following effects:

- The references from `file1.o` (`foo`, `MP3_Init()` and `MP3_Play()`) are matched to the definitions in `file2.o` (`bar`, `MyMP3_Init()` and `MyMP3_Play()` respectively), as specified by the steering file `ed.txt`.
- The `RESOLVE` command in `ed.txt` matches the `MP3` functions and the `--unresolved` option matches any other remaining references, in this case, `foo` to `bar`.
- The output symbol table, whether it is an image or a partial object, does not contain the symbols `foo`, `MP3_Init` or `MP3_Play`.

Related information

--edit=file_list on page 455
--unresolved=symbol on page 553
Edit the symbol tables with a steering file on page 639

3.9.7 SHOW steering file command

Makes global symbols visible.

The `SHOW` command is useful if you want to make a specific symbol visible that is hidden using a `HIDE` command with a wildcard.

Syntax

```
SHOW <pattern>[,<pattern>]
```

where:

`<pattern>`

is a string, optionally including wildcard characters, that matches zero or more global symbols. If `<pattern>` does not match any global symbol, the linker ignores the command.

Usage

The usage of `SHOW` is closely related to that of `HIDE`.

Related information

HIDE steering file command on page 761
Edit the symbol tables with a steering file on page 639

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4. fromelf Reference

A list of the command-line options for the `fromelf` command.

4.1 fromelf Command-line Options

Describes the command-line options of the `fromelf` image converter provided with Arm® Compiler for Embedded.

4.1.1 --base [[object_file::]load_region_ID=]num

Enables you to alter the base address specified for one or more load regions in Motorola S-record and Intel Hex file formats.

Syntax

```
--base [[<object_file>::]<load_region_ID>=]<num>
```

Parameters

`<object_file>`
An optional ELF input file.

`<load_region_ID>`
An optional load region. This can either be a symbolic name of an execution region belonging to a load region or a zero-based load region number, for example `#0` if referring to the first region.

`<num>`
Either a decimal or hexadecimal value.

Restrictions

Not supported for AArch64 state.

You must use one of the output formats `--i32`, `--i32combined`, `--m32`, or `--m32combined` with this option. Therefore, you cannot use this option with object files.

Operation

You can:

- Use wildcard characters `?` and `*` for symbolic names in `<object_file>` and `<load_region_ID>` arguments.
- Specify multiple values in one `<option>` followed by a comma-separated list of arguments.

All addresses encoded in the output file start at the base address `<num>`. If you do not specify a `--base` option, the base address is taken from the load region address.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Examples

Decimal value:

```
--base 0
```

Hexadecimal value:

```
--base 0x8000
```

Base address for the first load region:

```
--base #0=0
```

Base address for all load regions in `test.o`:

```
--base test.o::*=0
```

Base address for the first and second load regions:

```
--base #0=0,#1=0x8000
```

Related information

--i32 on page 802
--i32combined on page 803
--m32 on page 812
--m32combined on page 813
General considerations when using fromelf

4.1.2 --bin

Produces plain binary output, one file for each load region. You can split the output from this option into multiple files with the `--<width>x<banks>` option.

Syntax

```
--bin
```

Parameters

None.

Restrictions

The following restrictions apply:

- You cannot use this option with object files.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- You must use `--output` with this option.

Operation

Considerations when using `--bin`

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named `<destination>` and generates one binary output file for each load region in the input image. `fromelf` places the output files in the `<destination>` directory.


Note

For multiple load regions, the name of the first non-empty execution region in the corresponding load region is used for the filename.

A file is only created when the load region describes code or data that is present in the ELF file. For example a load region containing only execution regions with ZI data in them does not result in an output file.

Example: Convert an ELF file to a plain binary file

To convert an ELF file to a plain binary file, for example `outfile.bin`, enter:

```
fromelf --bin --output=outfile.bin infile.axf
```

Related information

--output=destination on page 815
--widthxbanks on page 834

4.1.3 --bincombined

Produces plain binary output. It generates one output file for an image containing multiple load regions.

Syntax

```
--bincombined
```

Parameters

None.

Restrictions

The following restrictions apply:

- You cannot use this option with object files.
- You must use `--output` with this option.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

By default, the start address of the first load region in memory is used as the base address. `fromelf` inserts padding between load regions as required to ensure that they are at the correct relative offset from each other. Separating the load regions in this way means that the output file can be loaded into memory and correctly aligned starting at the base address.

To change the default values for the base address and padding, use this option with `--bincombined_base` and `--bincombined_padding`.

Considerations when using `--bincombined`

Use this option with `--bincombined_base` to change the default value for the base address.

The default padding value is `0xFF`. Use this option with `--bincombined_padding` to change the default padding value.

If you use a scatter file that defines two load regions with a large gap in the address space between them, the resulting binary can be very large because it contains mostly padding. For example, if you have a load region of size `0x100` bytes at address `0x00000000` and another load region at address `0x30000000`, the amount of padding is `0x2FFFFF00` bytes.

Arm recommends that you use a different method of placing widely spaced load regions, such as `--bin`, and make your own arrangements to load the multiple output files at the correct addresses.

Example: Produce a binary file that can be loaded at start address 0x1000

```
fromelf --bincombined --bincombined_base=0x1000 --output=out.bin in.axf
```

Example: Produce plain binary output and fill the space between load regions with copies of a 32-bit word

Using the 32-bit word `0x12345678`, enter:

```
fromelf --bincombined --bincombined_padding=4,0x12345678 --output=out.bin in.axf
```

Related information

--bin on page 768
--bincombined_base=address on page 770
--bincombined_padding=size,num on page 771
--output=destination on page 815
--widthxbanks on page 834
Input sections, output sections, regions, and program segments on page 572

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.4 --bincombined_base=address

Enables you to lower the base address used by the `--bincombined` output mode. The output file generated is suitable to be loaded into memory starting at the specified address.

Default

By default the start address of the first load region in memory is used as the base address.

Syntax

```
--bincombined_base=<address>
```

Parameters

`<address>`

The start address where the image is to be loaded:

- If the specified address is lower than the start of the first load region, `fromelf` adds padding at the start of the output file.
- If the specified address is higher than the start of the first load region, `fromelf` gives an error.

Restrictions

You must use `--bincombined` with this option. If you omit `--bincombined`, a warning message is displayed.

Example: Specify the base address of the first load region as 0x1000

```
--bincombined --bincombined_base=0x1000
```

Related information

--bincombined on page 769
--bincombined_padding=size,num on page 771
Input sections, output sections, regions, and program segments on page 572

4.1.5 --bincombined_padding=size,num

Enables you to specify a different padding value from the default used by the `--bincombined` output mode.

Default

The default is `--bincombined_padding=1,0xFF`.

Syntax

```
--bincombined_padding=<size>,<num>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<size>

Is 1, 2, or 4 bytes to define whether it is a byte, halfword, or word.

<num>

The value to be used for padding. If you specify a value that is too large to fit in the specified size, a warning message is displayed.

Restrictions

You must use `--bincombined` with this option. If you omit `--bincombined`, a warning message is displayed.

Operation

`fromelf` expects that 2-byte and 4-byte padding values are specified in the appropriate endianness for the input file. For example, if you are translating a big endian ELF file into binary, the specified padding value is treated as a big endian word or halfword.

Example: Plain binary output and fill the space between load regions with copies of a 32-bit word

Using the 32-bit word `0x12345678`:

```
--bincombined --bincombined_padding=4,0x12345678
```

Example: Plain binary output and fill the space between load regions with copies of a 16-bit word

Using the 16-bit halfword `0x1234`:

```
--bincombined --bincombined_padding=2,0x1234
```

Example: For big endian memory, fill the space between load regions with 0x0100

```
--bincombined --bincombined_padding=2,0x01
```

Related information

--bincombined on page 769
--bincombined_base=address on page 770

4.1.6 --cad

Produces a C array definition or C++ array definition containing binary output.

Syntax

```
--cad
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Restrictions

You cannot use this option with object files.

Operation

You can use each array definition in the source code of another application. For example, you might want to embed an image in the address space of another application, such as an embedded operating system.

If your image has a single load region, the output is directed to `stdout` by default. To save the output to a file, use the `--output` option together with a filename.

If your image has multiple load regions, then you must also use the `--output` option together with a directory name. Unless you specify a full path name, the path is relative to the current directory. A file is created for each load region in the specified directory. The name of each file is the name of the corresponding execution region.

Use this option with `--output` to generate one output file for each load region in the image.

Considerations when using `--cad`

A file is only created when the load region describes code or data that is present in the ELF file. For example a load region containing only execution regions with ZI data in them does not result in an output file.

Example: Produce an array definition for an image that has a single load region

```
fromelf --cad myimage.axf

unsigned char LR0[] = {
    0x00,0x00,0x00,0xEB,0x28,0x00,0x00,0xEB,0x2C,0x00,0x8F,0xE2,0x00,0x0C,0x90,0xE8,
    0x00,0xA0,0x8A,0xE0,0x00,0xB0,0x8B,0xE0,0x01,0x70,0x4A,0xE2,0x0B,0x00,0x5A,0xE1,
    0x00,0x00,0x00,0x1A,0x20,0x00,0x00,0xEB,0x0F,0x00,0xBA,0xE8,0x18,0xE0,0x4F,0xE2,
    0x01,0x00,0x13,0xE3,0x03,0xF0,0x47,0x10,0x03,0xF0,0xA0,0xE1,0xAC,0x18,0x00,0x00,
    0xBC,0x18,0x00,0x00,0x00,0x30,0xB0,0xE3,0x00,0x40,0xB0,0xE3,0x00,0x50,0xB0,0xE3,
    0x00,0x60,0xB0,0xE3,0x10,0x20,0x52,0xE2,0x78,0x00,0xA1,0x28,0xFC,0xFF,0xFF,0x8A,
    0x82,0x2E,0xB0,0xE1,0x30,0x00,0xA1,0x28,0x00,0x30,0x81,0x45,0x0E,0xF0,0xA0,0xE1,
    0x70,0x00,0x51,0xE3,0x66,0x00,0x00,0x0A,0x64,0x00,0x51,0xE3,0x38,0x00,0x00,0x0A,
    0x00,0x00,0xB0,0xE3,0x0E,0xF0,0xA0,0xE1,0x1F,0x40,0x2D,0xE9,0x00,0x00,0xA0,0xE1,
.
.
.
    0x3A,0x74,0x74,0x00,0x43,0x6F,0x6E,0x73,0x74,0x72,0x75,0x63,0x74,0x65,0x64,0x20,
    0x41,0x20,0x23,0x25,0x64,0x20,0x61,0x74,0x20,0x25,0x70,0x0A,0x00,0x00,0x00,0x00,
    0x44,0x65,0x73,0x74,0x72,0x6F,0x79,0x65,0x64,0x20,0x41,0x20,0x23,0x25,0x64,0x20,
    0x61,0x74,0x20,0x25,0x70,0x0A,0x00,0x00,0x0C,0x99,0x00,0x00,0x0C,0x99,0x00,0x00,
    0x50,0x01,0x00,0x00,0x44,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
};
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: For an image that has multiple load regions, create a file for each load region in the directory

Create a file for each load region in the directory `<root>\myprojects\multiload\load_regions`. If `image_multiload.axf` contains the execution regions `EXEC_ROM` and `RAM`, then the files `EXEC_ROM` and `RAM` are created in the `load_regions` subdirectory:

```
cd <root>\myprojects\multiload
fromelf --cad image_multiload.axf --output load_regions
```

Related information

--cadcombined on page 774
--output=destination on page 815
Input sections, output sections, regions, and program segments on page 572

4.1.7 --cadcombined

Produces a C array definition or C++ array definition containing binary output.

Syntax

```
--cadcombined
```

Parameters

None.

Restrictions

You cannot use this option with object files.

Operation

You can use each array definition in the source code of another application. For example, you might want to embed an image in the address space of another application, such as an embedded operating system.

The output is directed to `stdout` by default. To save the output to a file, use the `--output` option together with a filename.

Example: Create a C or C++ array definition file in a directory

The following commands create the file `load_regions.c` in the directory `<root>\myprojects\multiload`:

```
cd root\myprojects\multiload
fromelf --cadcombined image_multiload.axf --output load_regions.c
```

Related information

--cad on page 772

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--output=destination on page 815

4.1.8 --compare=option[,option,...]

Compares two input files and prints the differences.

Syntax

```
--compare=<option>[,<option>,...]
```

Parameters

<option>

One of:

section_sizes

Compares the size of all sections for each ELF file or ELF member of a library file.

section_sizes::<object_name>

Compares the sizes of all sections in ELF objects with a name matching `<object_name>`.

section_sizes::<section_name>

Compares the sizes of all sections with a name matching `<section_name>`.

sections

Compares the size and contents of all sections for each ELF file or ELF member of a library file.

sections::<object_name>

Compares the size and contents of all sections in ELF objects with a name matching `<object_name>`.

sections::<section_name>

Compares the size and contents of all sections with a name matching `<section_name>`.

function_sizes

Compares the size of all functions for each ELF file or ELF member of a library file.

function_sizes::<object_name>

Compares the size of all functions in ELF objects with a name matching `<object_name>`.

function_size::<function_name>

Compares the size of all functions with a name matching `<function_name>`.

global_function_sizes

Compares the size of all global functions for each ELF file or ELF member of a library file.

global_function_sizes::<function_name>

Compares the size of all global functions in ELF objects with a name matching `<function_name>`.

You can:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Use wildcard characters `?` and `*` for symbolic names in `<section_name>`, `<function_name>`, and `<object_name>` arguments.
- Specify multiple values in one `<option>` followed by a comma-separated list of arguments.

Operation

The input files must be the same type, either two ELF files or two library files. Library files are compared member by member and the differences are concatenated in the output.

All differences between the two input files are reported as errors, unless they are downgraded to warnings by using the `--relax_section` option.

Related information

--ignore_section=option[,option,...] on page 804
--ignore_symbol=option[,option,...] on page 805
--relax_section=option[,option,...] on page 817
--relax_symbol=option[,option,...] on page 818

4.1.9 --continue_on_error

Reports any errors and then continues.

Syntax

```
--continue_on_error
```

Parameters

None.

Operation

Use `--diag_warning=error` instead of this option.

Related information

--diag_warning=tag[,tag,...] (fromelf) on page 788

4.1.10 --coprocN=value (fromelf)

Enables T32 encodings of the *Custom Datapath Extension* (CDE). These options are only compatible with M-profile targets and require the target to support at least Arm®v8-M mainline.

Syntax

```
--coproc<N>=<value>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<N>

The coprocessor ID in the range 0-7.

<value>

One of the following:

generic

This value is the default, and has the same effect as if `--coproc<N>` is omitted.

cde, CDE

Sets the instruction set architecture of the corresponding coprocessor encoding space to CDEv1.

Restrictions

You must use the `--cpu` option with `--coproc<N>=<value>`.

Example: CDE instructions with vector operands

The example is available in --coprocnvalue examples.

Related information

-march on page 123
-mcpu on page 150
--cpu=list (fromelf) on page 779

4.1.11 --coprocnvalue examples

This example shows how you can use the `fromelf` option `--coproc<N>=<value>`.

For more information about the `--coproc<N>=<value>` command-line option, see --coprocN=value (fromelf).

Example: CDE instructions with vector operands

1. Create the file `cde-vector.s` that uses coprocessor 0:

```
        .text
        .global f
f:
        vcx1a p0, q1, #1
        vcx1 p0, q2, #1

        vcx2a p0, q1, q2, #1
        vcx2 p0, q2, q3, #1

        vcx3a p0, q1, q2, q3, #1
        vcx3 p0, q1, q3, q4, #1
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

2. Compile cde-vector.S with:

```
armclang --target=arm-arm-none-eabi -march=armv8-m.main+cdecp0+mve -mfpu=fpv5-sp-
d16 -c cde-vector.S
```

3. Run `fromelf` to examine the generated assembly code:

```
fromelf -c --cpu=8-M.Main --coproc0=cde cde-vector.o

...
    f
        0x00000000:    fc202041     .A     VCX1A    p0,q1,#1
        0x00000004:    ec204041     .A@    VCX1     p0,q2,#1
        0x00000008:    fc302054    0.T     VCX2A    p0,q1,q2,#1
        0x0000000c:    ec304056    0.V@    VCX2     p0,q2,q3,#1
        0x00000010:    fc842056    ..V     VCX3A    p0,q1,q2,q3,#1
        0x00000014:    ec862058    ..X     VCX3     p0,q1,q3,q4,#1
...
```

If you do not specify a required feature, then the following errors are output:

- If you do not enable the CDE extension for a particular coprocessor, then you get errors such as:

```
armclang --target=arm-arm-none-eabi -march=armv8-m.main+mve -mfpu=fpv5-sp-d16 -c
 cde-vector.S

cde-vector.S:5:8: error: coprocessor must be configured as CDE
 vcx1a p0, q1, #1
...
```

- If you do not enable M-profile Vector Extension (MVE), then you get errors such as:

```
armclang --target=arm-arm-none-eabi -march=armv8-m.main+cdecp0 -mfpu=fpv5-sp-d16
 -c cde-vector.S

cde-vector.S:5:2: error: invalid instruction, any one of the following would fix
 this:
 vcx1a p0, q1, #1
 ^
cde-vector.S:5:12: note: operand must be a register in range [s0, s31]
 vcx1a p0, q1, #1
           ^
cde-vector.S:5:2: note: instruction requires: mve
 vcx1a p0, q1, #1
 ^
...
```

- If you do not enable an FPU with `-mfpu=none`, then you get an error for floating-point instructions, for example:

```
armclang --target=arm-arm-none-eabi -march=armv8-m.main -mfpu=none -c cde-
vector.S

cde-vector.S:5:2: error: invalid instruction
 vcx1a p0, q1, #1
 ^
...
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

Enabling MVE causes single-precision, double-precision, and vector registers to be available for CDE instructions, even if FPU is disabled.

- If you do not specify the `--coproc0=cde` option, then the disassmbly has `LDC` and `STC` instructions instead of the expected `VCX` instructions. For example:

```
fromelf -c --cpu=8-M.Main cde-vector.o

...
    f
        0x00000000:    fc202041     .A      STC2     p0,c2,[r0],#-0x104
        0x00000004:    ec204041     .A@     STC      p0,c4,[r0],#-0x104
        0x00000008:    fc302054    0.T      LDC2     p0,c2,[r0],#-0x150
        0x0000000c:    ec304056    0.V@     LDC      p0,c4,[r0],#-0x158
        0x00000010:    fc842056    ..V      STC2     p0,c2,[r4],<0x56>
        0x00000014:    ec862058    ..X      STC      p0,c2,[r6],<0x58>
```

4.1.12 --cpu=list (fromelf)

Lists the architecture and processor names that are supported by the `--cpu=name` option.

Note

Architectures and processors for Arm®v8.4-A or later are not listed because you cannot use `--cpu=<name>` for such targets.

Syntax

```
--cpu=list
```

Parameters

`list`

List the supported architecture and processor names.

Related information

--cpu=name (fromelf) on page 779

4.1.13 --cpu=name (fromelf)

Affects the way machine code is disassembled by options such as `-c` or `--disassemble`, so that it is disassembled in the same way that the specified processor or architecture interprets it.

Default

If you do not specify a `--cpu` option, then by default:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- For AArch32, `fromelf` disassembles all the instructions it knows about from the A- and R-profiles of the architecture. For M-profile architectures, you must specify the required architecture with `fromelf --cpu`.
- For AArch64, `fromelf` disassembles all the instructions it knows about from the A-profile of the architecture.


Note

To disassemble *M-profile Vector Extension* (MVE) instructions, you must specify the `--cpu` option for Arm®v8.1-M and later architectures. For example, `--cpu =8.1-M.Main.mve`.

To disassemble *Scalable Vector Extension* (SVE) instructions, you must not specify the `--cpu` option. `fromelf` cannot disassemble Armv8.4-A and later instructions without also disassembling SVE instructions.

Syntax

```
--cpu=<name>
```

Parameters

`<name>`

The name of a processor or architecture:

- Processor and architecture names are not case-sensitive.
- Wildcard characters are not accepted.

The following table shows the supported architectures. For a complete list of the supported architecture and processor names, specify the `--cpu=list` option.

Table 4-1: Supported Arm architectures

Architecture name	Description
`6-M`	Armv6 architecture microcontroller profile.
`6S-M`	Armv6 architecture microcontroller profile with OS extensions.
`7-A`	Armv7 architecture application profile.
`7-A.security`	Armv7-A architecture profile with the Security Extension and includes the `SMC` instruction (formerly `SMI`).
`7-R`	Armv7 architecture real-time profile.
`7-M`	Armv7 architecture microcontroller profile.
`7E-M`	Armv7-M architecture profile with DSP extension.
`8-A.32`	Armv8-A architecture profile, AArch32 state.
`8-A.32.crypto`	Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8-A.32.no_neon`	Armv8-A architecture profile, AArch32 state without Advanced SIMD instructions.
`8-A.64`	Armv8-A architecture profile, AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Architecture name	Description
`8-A.64.crypto`	Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.1-A.32`	Armv8.1, for Armv8-A architecture profile, AArch32 state.
`8.1-A.32.crypto`	Armv8.1, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.1-A.64`	Armv8.1, for Armv8-A architecture profile, AArch64 state.
`8.1-A.64.crypto`	Armv8.1, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.2-A.32`	Armv8.2, for Armv8-A architecture profile, AArch32 state.
`8.2-A.32.crypto`	Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.2-A.32.crypto.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions and the `VSDOT` and `VUDOT` instructions.
`8.2-A.32.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch32 state with the `VSDOT` and `VUDOT` instructions.
`8.2-A.64`	Armv8.2, for Armv8-A architecture profile, AArch64 state.
`8.2-A.64.crypto`	Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.2-A.64.crypto.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions and the `SDOT` and `UDOT` instructions.
`8.2-A.64.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch64 state with the `SDOT` and `UDOT` instructions.
`8.3-A.32`	Armv8.3, for Armv8-A architecture profile, AArch32 state.
`8.3-A.32.crypto`	Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.3-A.32.crypto.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions and the `VSDOT` and `VUDOT` instructions.
`8.3-A.32.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch32 state with the `VSDOT` and `VUDOT` instructions.
`8.3-A.64`	Armv8.3, for Armv8-A architecture profile, AArch64 state.
`8.3-A.64.crypto`	Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.3-A.64.crypto.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions and the `SDOT` and `UDOT` instructions.
`8.3-A.64.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch64 state with the `SDOT` and `UDOT` instructions.
`8-R`	Armv8-R architecture profile, AArch32 state.
`8-R.crypto`	Armv8-R architecture profile, AArch32 state with cryptographic instructions.
`8-R.no_neon`	Armv8-R architecture profile, AArch32 state without Advanced SIMD instructions.
`8-R.64`	Armv8-R architecture profile, AArch64 state.
`8-R.64.crypto`	Armv8-R architecture profile, AArch64 state with cryptographic instructions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Architecture name	Description
`8-M.Base`	Armv8-M baseline architecture profile. Derived from the Armv6-M architecture.
`8-M.Main`	Armv8-M mainline architecture profile. Derived from the Armv7-M architecture.
`8-M.Main.dsp`	Armv8-M mainline architecture profile with DSP extension.
`8.1-M.Main`	Armv8.1-M mainline architecture profile extension.
`8.1-M.Main.dsp`	Armv8.1-M mainline architecture profile with DSP extension.
`8.1-M.Main.mve`	Armv8.1-M mainline architecture profile with *M-profile Vector Extension* (MVE) for integer operations.
`8.1-M.Main.mve.fp`	Armv8.1-M mainline architecture profile with MVE for integer and floating-point operations.


Note

The full list of supported architectures and processors depends on your license.


Note

You cannot specify targets with Armv8.4-A or later architectures on the `fromelf` command line. To disassemble instructions for such targets, you must not specify the `--cpu` option when invoking `fromelf` directly.

Operation

The following general points apply to processor and architecture options:

Processors

Selecting the processor selects the appropriate architecture, *Floating-Point Unit* (FPU), and memory organization.

Architectures

If you specify an architecture name for the `--cpu` option, options such as `-c` or `--disassemble` disassemble machine code for that architecture. If you specify `--disassemble`, then the disassembly can be assembled for any processor supporting that architecture.

For example, `--cpu=7-A --disassemble` produces disassembly that can be assembled for the Cortex®-A7 processor.

FPU

- Some specifications of `--cpu` imply an `--fpu` selection.


Note

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

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- If no `--fpu` option is specified and no `--cpu` option is specified, then `fromelf` disassembles FPU instructions according to the default architecture settings.

Example: Specifying a processor

To specify the Cortex-M4 processor, use:

```
--cpu=Cortex-M4
```

Example: Effect of not specifying the correct -cpu option

Create the file `test.s` containing the following code:

```
    mrs x0, VSCTLR_EL2
    msr VSCTLR_EL2, x0
```

The system registers in this example use the same encoding as A-profile system registers but have different names in R-profile. Therefore, you must match the profile used in the compilation, and not rely on the default. The following example shows this difference.

Compile `test.s` with:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-r -c test.s
```

Run `fromelf` without `--cpu`. Because `8-A.64` is the default for AArch64, the disassembly is incorrect:

```
fromelf -c test.o

...
      0x00000000:    d53c2000    . <.    MRS      x0,TTBR0_EL2
      0x00000004:    d51c2000    . ..    MSR      TTBR0_EL2,x0
...
```

Run `fromelf` with `--cpu=8-R.64` to see the correct disassembly for Armv8-R AArch64:

```
fromelf --cpu=8-R.64 -c test.o

...
      0x00000000:    d53c2000    . <.    MRS      x0,VSCTLR_EL2
      0x00000004:    d51c2000    . ..    MSR      VSCTLR_EL2,x0
...
```

Related information

--cpu=list (fromelf) on page 779
--disassemble on page 789
--info=topic[,topic,...] (fromelf) on page 806
--text on page 826

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.14 --datasymbols

Modifies the output information of data sections so that symbol definitions are interleaved.

Syntax

```
--datasymbols
```

Parameters

None.

Operation

You can use this option only with `--text -d`.

Related information

--text on page 826

4.1.15 --debugonly

Removes the content of any code or data sections.

Syntax

```
--debugonly
```

Parameters

None.

Restrictions

You must use `--elf` with this option.

Operation

This option ensures that the output file contains only the information required for debugging, for example, debug sections, symbol table, and string table. Section headers are retained because they are required to act as targets for symbols.

Example: Create an ELF file containing only debug information from an input ELF file

To create an ELF file containing only debug information, `debugout.axf`, from the ELF file `infile.axf`, enter:

```
fromelf --elf --debugonly --output=debugout.axf infile.axf
```

Related information

--elf on page 791

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.16 --decode_build_attributes

Prints the contents of the build attributes section in human-readable form for standard build attributes or raw hexadecimal form for nonstandard build attributes.


Note

The standard build attributes are documented in the *Application Binary Interface for the Arm Architecture*.

Syntax

```
--decode_build_attributes
```

Parameters

None.

Restrictions

This option has no effect for AArch64 state inputs.

Example

The following example shows the output for `--decode_build_attributes`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c hello.c -o hello.o
fromelf -v --decode_build_attributes hello.o

...
** Section #6

    Name         : .ARM.attributes
    Type         : SHT_ARM_ATTRIBUTES (0x70000003)
    Flags        : None (0x00000000)
    Addr         : 0x00000000
    File Offset  : 112 (0x70)
    Size         : 74 bytes (0x4a)
    Link         : SHN_UNDEF
    Info         : 0
    Alignment    : 1
    Entry Size   : 0

    'aeabi' file build attributes:
    0x000000:   43 32 2e 30 39 00 05 63 6f 72 74 65 78 2d 61 35    C2.09..cortex-a5
    0x000010:   33 00 06 0e 07 41 08 01 09 02 0a 07 0c 03 0e 00    3....A..........
    0x000020:   11 01 12 04 14 01 15 01 17 03 18 01 19 01 1a 02    ................
    0x000030:   22 00 24 01 26 01 2a 01 44 03                      ".$.&.*.D.
        Tag_conformance = "2.09"
        Tag_CPU_name = "cortex-a53"
        Tag_CPU_arch = ARM v8 (=14)
        Tag_CPU_arch_profile = The application profile 'A' (e.g. for Cortex A8)
(=65)
        Tag_ARM_ISA_use = ARM instructions were permitted to be used (=1)
        Tag_THUMB_ISA_use = Thumb2 instructions were permitted (implies Thumb in
structions permitted) (=2)
        Tag_VFP_arch = Use of the ARM v8-A FP ISA was permitted (=7)
        Tag_NEON_arch = Use of the ARM v8-A Advanced SIMD Architecture (Neon) wa
s permitted (=3)
        Tag_ABI_PCS_R9_use = R9 used as V6 (just another callee-saved register)
(=0)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
        Tag_ABI_PCS_GOT_use = Data are imported directly (=1)
        Tag_ABI_PCS_wchar_t = Size of wchar_t is 4 (=4)
        Tag_ABI_FP_denormal = This code was permitted to require IEEE 754 denorm
al numbers (=1)
        Tag_ABI_FP_exceptions = This code was permitted to check the IEEE 754 in
exact exception (=1)
        Tag_ABI_FP_number_model = This code may use all the IEEE 754-defined FP
encodings (=3)
        Tag_ABI_align8_needed = Code was permitted to depend on the 8-byte align
ment of 8-byte data items (=1)
        Tag_ABI_align8_preserved = Code was required to preserve 8-byte alignmen
t of 8-byte data objects (=1)
        Tag_ABI_enum_size = Enum containers are 32-bit (=2)
        Tag_CPU_unaligned_access = The producer was not permitted to make unalig
ned data accesses (=0)
        Tag_VFP_HP_extension = The producer was permitted to use the VFPv3/Advan
ced SIMD optional half-precision extension (=1)
        Tag_ABI_FP_16bit_format = The producer was permitted to use IEEE 754 for
mat 16-bit floating point numbers (=1)
        Tag_MPextension_use = Use of the ARM v7 MP extension was permitted (=1)
        Tag_Virtualization_use = Use of TrustZone and virtualization extensions
was permitted (=3)
...
```

Related information

--dump_build_attributes on page 790
--emit=option[,option,...] on page 791
--extract_build_attributes on page 795
Application Binary Interface for the Arm Architecture

4.1.17 --diag_error=tag[,tag,...] (fromelf)

Sets diagnostic messages that have a specific tag to Error severity.

Syntax

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

Parameters

<tag>

One of:

- 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.

Related information

--diag_remark=tag[,tag,...] (fromelf) on page 786
--diag_style=arm|ide|gnu (fromelf) on page 787
--diag_suppress=tag[,tag,...] (fromelf) on page 788
--diag_warning=tag[,tag,...] (fromelf) on page 788

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.18 --diag_remark=tag[,tag,...] (fromelf)

Sets diagnostic messages that have a specific tag to Remark severity.

Syntax

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

Parameters

<tag>

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 information

--diag_error=tag[,tag,...] (fromelf) on page 786
--diag_style=arm|ide|gnu (fromelf) on page 787
--diag_suppress=tag[,tag,...] (fromelf) on page 788
--diag_warning=tag[,tag,...] (fromelf) on page 788

4.1.19 --diag_style=arm|ide|gnu (fromelf)

Specifies the display style for diagnostic messages.

Default

The default is `--diag_style=arm`.

Syntax

```
--diag_style=<string>
```

Parameters

<string>

One of:

arm

Display messages using the legacy 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`.

Operation

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

Related information

--diag_error=tag[,tag,...] (fromelf) on page 786
--diag_remark=tag[,tag,...] (fromelf) on page 786
--diag_suppress=tag[,tag,...] (fromelf) on page 788
--diag_warning=tag[,tag,...] (fromelf) on page 788

4.1.20 --diag_suppress=tag[,tag,...] (fromelf)

Suppresses diagnostic messages that have a specific tag.

Syntax

```
--diag_suppress=<tag>[,<tag>,...]
```

Parameters

`<tag>`

One of:

- 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 information

--diag_error=tag[,tag,...] (fromelf) on page 786
--diag_remark=tag[,tag,...] (fromelf) on page 786
--diag_style=arm|ide|gnu (fromelf) on page 787
--diag_warning=tag[,tag,...] (fromelf) on page 788

4.1.21 --diag_warning=tag[,tag,...] (fromelf)

Sets diagnostic messages that have a specific tag to Warning severity.

Syntax

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

Parameters

`<tag>`

One of:

- 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.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- `error`, to set all errors that can be downgraded to warnings.

Related information

--diag_error=tag[,tag,...] (fromelf) on page 786
--diag_remark=tag[,tag,...] (fromelf) on page 786
--diag_style=arm|ide|gnu (fromelf) on page 787

4.1.22 --disassemble

Displays a disassembled version of the image to `stdout`. Disassembly is generated in `armasm` assembler syntax and not GNU assembler syntax.

Syntax

```
--disassemble
```

Parameters

None.

Operation

If you use this option with `--output <destination>`, you can reassemble the output file with `armasm`.

You can use this option to disassemble either an ELF image or an ELF object file.


Note

The output is not the same as that from `--emit=code` and `--text -c`.


Note

To disassemble *M-profile Vector Extension* (MVE) instructions, you must specify the `--cpu` option for Arm®v8.1-M and later architectures. For example, `--cpu =8.1-M.Main.mve`.

To disassemble *Scalable Vector Extension* (SVE) instructions, you must not specify the `--cpu` option. `fromelf` cannot disassemble Armv8.4-A and later instructions without also disassembling Scalable Vector Extension SVE instructions.

`armasm` does not support:

- Armv8.4-A and later architectures.
- Certain backported options in Armv8.2-A and Armv8.3-A.
- Assembling MVE or SVE instructions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

To disassemble the ELF file infile.axf for the Cortex®-A7 processor and create a source file `outfile.asm`, enter:

```
fromelf --cpu=Cortex-A7 --disassemble --output=outfile.asm infile.axf
```

Related information

--cpu=name (fromelf) on page 779
--emit=option[,option,...] on page 791
--interleave=option on page 809
--output=destination on page 815
--text on page 826

4.1.23 --dump_build_attributes

Prints the contents of the build attributes section in raw hexadecimal form.

Syntax

```
--dump_build_attributes
```

Parameters

None.

Restrictions

This option has no effect for AArch64 state inputs.

Example

The following example shows the output for `--dump_build_attributes`:

```
...

** Section #10 '.ARM.attributes' (SHT_ARM_ATTRIBUTES)
    Size   : 89 bytes

    0x000000:   41 47 00 00 00 61 65 61 62 69 00 01 3d 00 00 00    AG...aeabi..=...
    0x000010:   43 32 2e 30 36 00 05 38 2d 41 2e 33 32 00 06 0a    C2.06..8-A.32...
    0x000020:   07 41 08 01 09 02 0a 05 0c 02 11 01 12 02 14 02    .A..............
    0x000030:   17 01 18 01 19 01 1a 01 1c 01 1e 03 22 01 24 01    ............".$.
    0x000040:   42 01 44 03 46 01 2c 02 11 00 00 00 41 52 4d 00    B.D.F.,.....ARM.
    0x000050:   01 09 00 00 00 12 01 16 01                         .........
```

Related information

--decode_build_attributes on page 784
--emit=option[,option,...] on page 791
--extract_build_attributes on page 795
--text on page 826

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.24 --elf

Selects ELF output mode.

Syntax

```
--elf
```

Parameters

None.

Restrictions

You must use `--output` with this option. For more information, see --output=destination.

Operation

Use this option whenever you have to transform an ELF file into a slightly different ELF file. You also have to provide options to indicate how you want the file to be modified. The options are:

- --debugonly.
- --globalize=option[,option,...].
- --hide=option[,option,...].
- --hide_and_localize=option[,option,...].
- --in_place.
- --linkview, --no_linkview. This option is deprecated.
- --localize=option[,option,...].
- --rename=option[,option,...].
- --show=option[,option,...].
- --show_and_globalize=option[,option,...].
- --strip=option[,option,...].
- --symbolversions, --no_symbolversions.

4.1.25 --emit=option[,option,...]

Enables you to specify the elements of an ELF object that you want to appear in the textual output. The output includes ELF header and section information.

Syntax

```
--emit=<option>[,<option>,...]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<option>

One of:

addresses

Prints global and static data addresses, including addresses for structure and union contents. It has the same effect as `--text -a`.

This option can only be used on files containing debug information. If no debug information is present, a warning message is generated.

Use the `--select` option to output a subset of the data addresses.

If you want to view the data addresses of arrays, expanded both inside and outside structures, use the `--expandarrays` option with this text category.

build_attributes

Prints the contents of the build attributes section in human-readable form for standard build attributes or raw hexadecimal form for nonstandard build attributes. The produces the same output as the `--decode_build_attributes` option.

code

Disassembles code, alongside a dump of the original binary data being disassembled and the addresses of the instructions. It has the same effect as `--text -c`.


Note

Unlike the output from `--disassemble`, the disassembly cannot be input to the assembler.

data

Prints contents of the data sections. It has the same effect as `--text -d`.

data_symbols

Modifies the output information of data sections so that symbol definitions are interleaved.

debug_info

Prints debug information. It has the same effect as `--text -g`.

dynamic_segment

Prints dynamic segment contents. It has the same effect as `--text -y`.

exception_tables

Decodes AArch32 exception table information for objects. It has the same effect as `--text -e`.

frame_directives

Prints the contents of `FRAME` directives in disassembled code as specified by the debug information embedded in an object module.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Use this option with `--disassemble`.

`heading_comments`
Prints heading comments at the beginning of the disassembly containing tool and command-line information from `.comment` sections.

Use this option with `--disassemble`.

`raw_build_attributes`
Prints the contents of the build attributes section in raw hexadecimal form, that is, in the same form as data.

`relocation_tables`
Prints relocation information. It has the same effect as `--text -r`.

`string_tables`
Prints the string tables. It has the same effect as `--text -t`.

`summary`
Prints a summary of the segments and sections in a file. It is the default output of `fromelf --text`. However, the summary is suppressed by some `--info` options. Use `--emit summary` to explicitly re-enable the summary, if required.

`symbol_annotations`
Prints symbols in disassembled code and data annotated with comments containing the respective property information.

Use this option with `--disassemble`.

`symbol_tables`
Prints the symbol and versioning tables. It has the same effect as `--text -s`.

`whole_segments`
Prints disassembled executables or shared libraries segment by segment even if it has a link view.

Use this option with `--disassemble`.

You can specify multiple options in one `<option>` followed by a comma-separated list of arguments.

Restrictions

You can use this option only in text mode.

Related information

--disassemble on page 789
--decode_build_attributes on page 784
--expandarrays on page 793
--text on page 826

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.26 --expandarrays

Prints data addresses, including arrays that are expanded both inside and outside structures.

Syntax

```
--expandarrays
```

Parameters

None.

Restrictions

You can use this option with `--text -a` or with `--fieldoffsets`.

Example: Output for a struct containing arrays when specifying -fieldoffsets and -expandarrays

The following example shows the output for a struct containing arrays when you specify both `--fieldoffsets` and `--expandarrays`:

```
// foo.c
struct S {
    char A[8];
    char B[4];
};
struct S s;

struct S* get()
{
    return &s;
}
```

```
>  armclang -target arm-arm-none-eabi -march=armv8-a -g -c foo.c

> fromelf --fieldoffsets --expandarrays foo.o
```

```
; Structure, S , Size 0xc bytes, from foo.c

|S.A|                                    EQU    0        ;  array[8] of char
|S.A[0]|                                 EQU    0        ;  char
|S.A[1]|                                 EQU    0x1      ;  char
|S.A[2]|                                 EQU    0x2      ;  char
|S.A[3]|                                 EQU    0x3      ;  char
|S.A[4]|                                 EQU    0x4      ;  char
|S.A[5]|                                 EQU    0x5      ;  char
|S.A[6]|                                 EQU    0x6      ;  char
|S.A[7]|                                 EQU    0x7      ;  char
|S.B|                                    EQU    0x8      ;  array[4] of char
|S.B[0]|                                 EQU    0x8      ;  char
|S.B[1]|                                 EQU    0x9      ;  char
|S.B[2]|                                 EQU    0xa      ;  char
|S.B[3]|                                 EQU    0xb      ;  char
; End of Structure S

    END
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--fieldoffsets on page 796
--text on page 826

4.1.27 --extract_build_attributes

Prints only the build attributes in a form that depends on the type of attribute.

Syntax

```
--extract_build_attributes
```

Parameters

None.

Restrictions

This option has no effect for AArch64 state inputs.

Operation

Prints the build attributes in:

- Human-readable form for standard build attributes.
- Raw hexadecimal form for nonstandard build attributes.

Example

The following example shows the output for `--extract_build_attributes`:

```
> armclang -c -mcpu=cortex-m7 --target=arm-arm-none-eabi -mfpu=vfpv3 hello.c -o
 hello.o
> fromelf --cpu=Cortex-M7 --extract_build_attributes hello.o

==============================================================================

** Object/Image Build Attributes

    'aeabi' file build attributes:
    0x000000:   43 32 2e 30 39 00 05 63 6f 72 74 65 78 2d 6d 37    C2.09..cortex-m7
    0x000010:   00 06 0d 07 4d 08 00 09 02 0a 05 0e 00 11 01 12    ....M...........
    0x000020:   04 14 01 15 01 17 03 18 01 19 01 1a 02 22 00 24    .............".$
    0x000030:   01 26 01                                           .&.
        Tag_conformance = "2.09"
        Tag_CPU_name = "cortex-m7"
        Tag_CPU_arch = ARM v7E-M (=13)
        Tag_CPU_arch_profile = The microcontroller profile 'M' (e.g. for Cortex M3)
 (=77)
        Tag_ARM_ISA_use = No ARM instructions were permitted to be used (=0)
        Tag_THUMB_ISA_use = Thumb2 instructions were permitted (implies Thumb
 instructions permitted) (=2)
        Tag_VFP_arch = VFPv4 instructions were permitted (implies VFPv3 instructions
 were permitted) (=5)
        Tag_ABI_PCS_R9_use = R9 used as V6 (just another callee-saved register) (=0)
        Tag_ABI_PCS_GOT_use = Data are imported directly (=1)
        Tag_ABI_PCS_wchar_t = Size of wchar_t is 4 (=4)
        Tag_ABI_FP_denormal = This code was permitted to require IEEE 754 denormal
 numbers (=1)
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
        Tag_ABI_FP_exceptions = This code was permitted to check the IEEE 754
inexact exception (=1)
        Tag_ABI_FP_number_model = This code may use all the IEEE 754-defined FP
encodings (=3)
        Tag_ABI_align8_needed = Code was permitted to depend on the 8-byte alignment
of 8-byte data items (=1)
        Tag_ABI_align8_preserved = Code was required to preserve 8-byte alignment of
8-byte data objects (=1)
        Tag_ABI_enum_size = Enum containers are 32-bit (=2)
        Tag_CPU_unaligned_access = The producer was not permitted to make unaligned
data accesses (=0)
        Tag_VFP_HP_extension = The producer was permitted to use the VFPv3/Advanced
SIMD optional half-precision extension (=1)
        Tag_ABI_FP_16bit_format = The producer was permitted to use IEEE 754 format
16-bit floating point numbers (=1)
```

Related information

--decode_build_attributes on page 784
--dump_build_attributes on page 790
--emit=option[,option,...] on page 791
--text on page 826

4.1.28 --fieldoffsets

Prints a list of `armasm` style assembly language `EQU` directives that equate C++ class or C structure field names to their offsets from the base of the class or structure.

Syntax

```
--fieldoffsets
```

Parameters

None.

Restrictions

This option:

- Requires that the object or image file has debug information.
- Can be used in text mode and with `--expandarrays`.

Operation

The input ELF file can be a relocatable object or an image.

Use `--output` to redirect the output to a file. Use the `INCLUDE` directive from `armasm` to load the produced file and provide access to C++ classes and C structure members by name from assembly language.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

The `EQU` directives cannot be used with the armclang integrated assembler. To use them, you must change them to GNU syntax, as described in Miscellaneous directives in the *Arm Compiler for Embedded Migration and Compatibility Guide*.

This option outputs all structure information. To output a subset of the structures, use `--select <select_options>`.

If you do not require a file that can be input to `armasm`, use the `--text -a` options to format the display addresses in a more readable form. The `-a` option only outputs address information for structures and static data in images because the addresses are not known in a relocatable object.

Example: Produce an output listing to stdout that contains all the field offsets from all structures

To produce an output listing in the file `inputfile.o`, enter:

```
fromelf --fieldoffsets inputfile.o
```

Example: Produce an output listing to a file that contains all the field offsets from structures with a name starting with p

To produce an output file listing to `outputfile.s` that contains all the field offsets from structures in the file `inputfile.o` that have a name starting with `p`, enter:

```
fromelf --fieldoffsets --select=p* --output=outputfile.s inputfile.o
```

Example: Produce an output listing to a file that contains all the field offsets from structures with names of tools or moretools

To produce an output listing to `outputfile.s` that contains all the field offsets from structures in the file `inputfile.o` with names of `tools` or `moretools`, enter:

```
fromelf --fieldoffsets --select=tools.*,moretools.* --output=outputfile.s
 inputfile.o
```

Example: Produce an output listing to a file containing all field offsets of structure fields with names starting with number and are within structure field top in structure tools

To produce an output file listing to `outputfile.s` that contains all the field offsets of structure fields whose name starts with `number` and are within structure field `top` in structure `tools` in the file `inputfile.o`, enter:

```
fromelf --fieldoffsets --select=tools.top.number* --output=outputfile.s inputfile.o
```

The following is an example of the output, and includes `name.` and `name...member` that arise because of anonymous structs and unions:

```
; Structure, Table , Size 0x104 bytes, from inputfile.cpp
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
|Table.TableSize|                          EQU    0        ;  int
|Table.Data|                               EQU    0x4      ;  array[64] of
 MyClassHandle
; End of Structure Table
; Structure, Box2 , Size 0x8 bytes, from inputfile.cpp
|Box2.|                                   EQU    0        ;  anonymous
|Box2..|                                  EQU    0        ;  anonymous
|Box2...Min|                              EQU    0        ;  Point2
|Box2...Min.x|                            EQU    0        ;  short
|Box2...Min.y|                            EQU    0x2      ;  short
|Box2...Max|                              EQU    0x4      ;  Point2
|Box2...Max.x|                            EQU    0x4      ;  short
|Box2...Max.y|                            EQU    0x6      ;  short
; Warning: duplicate name (Box2..) present in (inputfile.cpp) and in (inputfile.cpp)
; please use the --qualify option
|Box2..|                                  EQU    0        ;  anonymous
|Box2...Left|                             EQU    0        ;  unsigned short
|Box2...Top|                              EQU    0x2      ;  unsigned short
|Box2...Right|                            EQU    0x4      ;  unsigned short
|Box2...Bottom|                           EQU    0x6      ;  unsigned short
; End of Structure Box2
; Structure, MyClassHandle , Size 0x4 bytes, from inputfile.cpp
|MyClassHandle.Handle|                    EQU    0        ;  pointer to MyClass
; End of Structure MyClassHandle
; Structure, Point2 , Size 0x4 bytes, from defects.cpp
|Point2.x|                                EQU    0        ;  short
|Point2.y|                                EQU    0x2      ;  short
; End of Structure Point2
; Structure, __fpos_t_struct , Size 0x10 bytes, from <filepath>
|__fpos_t_struct.__pos|                   EQU    0        ;  unsigned long long
|__fpos_t_struct.__mbstate|               EQU    0x8      ;  anonymous
|__fpos_t_struct.__mbstate.__state1|      EQU    0x8      ;  unsigned int
|__fpos_t_struct.__mbstate.__state2|      EQU    0xc      ;  unsigned int
; End of Structure __fpos_t_struct
    END
```

Related information

--expandarrays on page 793
--qualify on page 816
--select=select_options on page 819
--text on page 826
EQU directive on page 992
GET or INCLUDE directive on page 1007
Miscellaneous directives

4.1.29 --fpu=list (fromelf)

Lists the *Floating Point Unit* (FPU) architectures that are supported by the `--fpu=name` option. Deprecated options are not listed.

Syntax

```
--fpu=list
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

`list`
List of supported FPU architectures.

Related information

--fpu=name (fromelf) on page 799

4.1.30 --fpu=name (fromelf)

Specifies the target FPU architecture.

Default

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

If the CPU you specify with `--cpu` has a VFP coprocessor, the default target FPU architecture is the VFP architecture for that CPU.

Syntax

```
--fpu=<name>
```

Parameters

`<name>`
The name of the target FPU architecture. Specify `--fpu=list` to list the supported FPU architecture names that you can use with `--fpu=name`.

The default floating-point architecture depends on the target architecture.

Restrictions

Software floating-point linkage is not available for AArch64 state.

Operation

This option selects disassembly for a specific FPU architecture. It affects how fromelf interprets the instructions it finds in the input files.

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.

Any FPU explicitly selected using the `--fpu` option always overrides any FPU implicitly selected using the `--cpu` option.

Related information

--disassemble on page 789
--fpu=list (fromelf) on page 798
--info=topic[,topic,...] (fromelf) on page 806

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--text on page 826

4.1.31 --globalize=option[,option,...]

Converts the selected symbols to global symbols.

Syntax

```
--globalize=<option>[,<option>,...]
```

Parameters

<option>

One of:

<object_name>::

All symbols in ELF objects with a name matching <object_name> are converted to global symbols.

<object_name>::<symbol_name>

All symbols in ELF objects with a name matching <object_name> and also a symbol name matching <symbol_name> are converted to global symbols.

<symbol_name>

All symbols with a symbol name matching <symbol_name> are converted to global symbols.

You can:

- Use wildcard characters ? and * for symbolic names in <symbol_name> and <object_name> arguments
- Specify multiple values in one <option> followed by a comma-separated list of arguments.

Restrictions

You must use --elf with this option.

Related information

--elf on page 791
--hide=option[,option,...] on page 801

4.1.32 --help (fromelf)

Displays a summary of the main command-line options.

Default

This option is the default if you specify the tool command without any options or source files.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
--help
```

Parameters

None.

Related information

--show_cmdline (fromelf) on page 822
--version_number (fromelf) on page 829
--vsn (fromelf) on page 831

4.1.33 --hide=option[,option,...]

Changes the symbol visibility property to mark selected symbols as hidden.

Syntax

```
--hide=<option>[,<option> ,...]
```

Parameters

<option>

One of:

<object_name>::

All symbols in ELF objects with a name matching <object_name>.

<object_name>::<symbol_name>

All symbols in ELF objects with a name matching <object_name> and also a symbol name matching <symbol_name>.

<symbol_name>

All symbols with a symbol name matching <symbol_name>.

You can:

- Use wildcard characters ? and * for symbolic names in <symbol_name> and <object_name> arguments
- Specify multiple values in one <option> followed by a comma-separated list of arguments.

Restrictions

You must use --elf with this option.

Related information

--elf on page 791
--show=option[,option,...] on page 821

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.34 --hide_and_localize=option[,option,...]

Changes the symbol visibility property to mark selected symbols as hidden, and converts the selected symbols to local symbols.

Syntax

```
--hide_and_localize=<option>[,<option>,...]
```

Parameters

<option>

One of:

<object_name>::

All symbols in ELF objects with a name matching <object_name> are marked as hidden and converted to local symbols.

<object_name>::<symbol_name>

All symbols in ELF objects with a name matching <object_name> and also a symbol name matching <symbol_name> are marked as hidden and converted to local symbols.

<symbol_name>

All symbols with a symbol name matching <symbol_name> are marked as hidden and converted to local symbols.

You can:

- Use wildcard characters ? and * for symbolic names in <symbol_name> and <object_name> arguments
- Specify multiple values in one <option> followed by a comma-separated list of arguments.

Restrictions

You must use --elf with this option.

Related information

--elf on page 791

4.1.35 --i32

Produces Intel Hex-32 format output. It generates one output file for each load region in the image.

You can specify the base address of the output with the --base option.

Syntax

```
--i32
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Restrictions

The following restrictions apply:

- Not supported for AArch64 state.
- You cannot use this option with object files.
- You must use `--output` with this option.

Operation

Considerations when using `--i32`

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named `<destination>` and generates one binary output file for each load region in the input image. `fromelf` places the output files in the `<destination>` directory.



For multiple load regions, the name of the first non-empty execution region in the corresponding load region is used for the filename.

A file is only created when the load region describes code or data that is present in the ELF file. For example a load region containing only execution regions with ZI data in them does not result in an output file.

Example: Convert an ELF file to an Intel Hex-32 format file

To convert the ELF file `infile.axf` to an Intel Hex-32 format file `outfile.bin`, enter:

```
fromelf --i32 --output=outfile.bin infile.axf
```

Related information

--base [[object_file::]load_region_ID=]num on page 767
--i32combined on page 803
--output=destination on page 815

4.1.36 --i32combined

Produces Intel Hex-32 format output. It generates one output file for an image containing multiple load regions.

You can specify the base address of the output with the `--base` option.

Syntax

```
--i32combined
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Restrictions

The following restrictions apply:

- Not supported for AArch64 state.
- You cannot use this option with object files.
- You must use `--output` with this option.

Operation

Considerations when using `--i32combined`

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named `<destination>` and generates one binary output file for all load regions in the input image. `fromelf` places the output file in the `<destination>` directory.

ELF images contain multiple load regions if, for example, they are built with a scatter file that defines more than one load region.

Example

To create a single output file, `outfile2.bin`, from an image file `infile2.axf`, with two load regions, and with a start address of `0x1000`, enter:

```
fromelf --i32combined --base=0x1000 --output=outfile2.bin infile2.axf
```

Related information

--base [[object_file::]load_region_ID=]num on page 767
--i32 on page 802
--output=destination on page 815

4.1.37 --ignore_section=option[,option,...]

Specifies the sections to be ignored during a compare. Differences between the input files being compared are ignored if they are in these sections.

Syntax

```
--ignore_section=<option>[,<option>,...]
```

Parameters

`<option>`

One of:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<object_name>::
All sections in ELF objects with a name matching <object_name>.

<object_name>::<section_name>
All sections in ELF objects with a name matching <object_name> and also a section name matching <section_name>.

<section_name>
All sections with a name matching <section_name>.

You can:

- Use wildcard characters ? and * for symbolic names in <symbol_name> and <object_name> arguments
- Specify multiple values in one <option> followed by a comma-separated list of arguments.

Restrictions

You must use --compare with this option.

Related information

--compare=option[,option,...] on page 775
--ignore_symbol=option[,option,...] on page 805
--relax_section=option[,option,...] on page 817

4.1.38 --ignore_symbol=option[,option,...]

Specifies the symbols to be ignored during a compare. Differences between the input files being compared are ignored if they are related to these symbols.

Syntax

```
--ignore_symbol=<option>[,<option>,...]
```

Parameters

<option>
One of:

<object_name>::
All symbols in ELF objects with a name matching <object_name>.

<object_name>::<symbol_name>
All symbols in ELF objects with a name matching <object_name> and also all symbols with names matching <symbol_name>.

<symbol_name>
All symbols with names matching <symbol_name>.

You can:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Use wildcard characters `?` and `*` for symbolic names in `<symbol_name>` and `<object_name>` arguments
- Specify multiple values in one `<option>` followed by a comma-separated list of arguments.

Restrictions

You must use `--compare` with this option.

Related information

--compare=option[,option,...] on page 775
--ignore_section=option[,option,...] on page 804
--relax_symbol=option[,option,...] on page 818

4.1.39 --in_place

Enables the translation of ELF members in an input file to overwrite the previous content.

Syntax

```
--in_place
```

Parameters

None.

Restrictions

You must use `--elf` with this option.

Example: Remove debug information from members of the library file

To remove debug information from members of the library file `test.a`, enter:

```
fromelf --elf --in_place --strip=debug test.a
```

Related information

--elf on page 791
--strip=option[,option,...] on page 823

4.1.40 --info=topic[,topic,...] (fromelf)

Prints information about specific topics.

Syntax

```
--info=<topic>[,<topic>,...]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<topic>

A comma-separated list from the following topic keywords:

instruction_usage

Categorizes and lists the A32 and T32 instructions defined in the code sections of each input file.


Note

Not supported for AArch64 state.

function_sizes

Lists the names of the global functions defined in one or more input files, together with their sizes in bytes and whether they are A32 or T32 functions.

function_sizes_all

Lists the names of the local and global functions defined in one or more input files, together with their sizes in bytes and whether they are A32 or T32 functions.

sizes

Lists the `Code`, `RO Data`, `RW Data`, `ZI Data`, and `Debug` sizes for each input object and library member in the image. Using this option implies `--info=sizes,totals`.

totals

Lists the totals of the `Code`, `RO Data`, `RW Data`, `ZI Data`, and `Debug` sizes for input objects and libraries.


Note

Code related sizes also include the size of any execute-only code.

The output from `--info=sizes,totals` always includes the padding values in the totals for input objects and libraries.


Note

Spaces are not permitted between topic keywords in the list. For example, you can enter `--info=sizes,totals` but not `--info=sizes, totals`.

Restrictions

`instruction_usage` is mot supported for AArch64 state.

You can use this option only in text mode.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--text on page 826

4.1.41 input_file (fromelf)

Specifies the ELF file or archive containing ELF files to be processed.

Operation

Multiple input files are supported if you:

- Output `--text` format.
- Use the `--compare` option.
- Use `--elf` with `--in_place`.
- Specify an output directory using `--output`.

If `<input_file>` is a scatter-loaded image that contains more than one load region and the output format is one of `--bin` , `--cad`, `--m32`, `--i32`, or `--vhx`, then `fromelf` creates a separate file for each load region.

If `<input_file>` is a scatter-loaded image that contains more than one load region and the output format is one of `--cadcombined` , `--m32combined`, or `--i32combined`, then `fromelf` creates a single file containing all load regions.

If `<input_file>` is an archive, you can process all files, or a subset of files, in that archive. To process a subset of files in the archive, specify a filter after the archive name as follows:

```
archive.a(filter_pattern)
```

Where `<filter_pattern>` specifies a member file. To specify a subset of files use the following wildcard characters:

*****

Matches zero or more characters.

?

Matches any single character.



Note

On Unix systems your shell typically requires the parentheses and these characters to be escaped with backslashes. Alternatively, enclose the archive name and filter in single quotes. For example, either of the following would work:

```
archive.a\(\?\?str\*\)
'archive.a(??str*)'
```

Any files in the archive that are not processed are included in the output archive together with the processed files.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

To convert all files in the archive beginning with `s`, and create a new archive, `my_archive.a`, containing the processed and unprocessed files, enter:

```
fromelf archive.a(s*.o) --output=my_archive.a
```

Related information

--bin on page 768
--cad on page 772
--cadcombined on page 774
--compare=option[,option,...] on page 775
--elf on page 791
--i32 on page 802
--i32combined on page 803
--in_place on page 806
--m32 on page 812
--m32combined on page 813
--output=destination on page 815
--text on page 826
--vhx on page 829
Examples of processing ELF files in an archive

4.1.42 --interleave=option

Inserts the original source code as comments into the disassembly if debug information is present.

Default

The default is `--interleave=none`.

Syntax

```
--interleave=<option>
```

Parameters

`<option>`

One of the following:

`line_directives`

Interleaves `#line` directives containing filenames and line numbers of the disassembled instructions.

`line_numbers`

Interleaves comments containing filenames and line numbers of the disassembled instructions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

none

Disables interleaving. This is useful if you have a generated makefile where the fromelf command has multiple options in addition to `--interleave`. You can then specify `--interleave=none` as the last option to ensure that interleaving is disabled without having to reproduce the complete fromelf command.

Operation

Use this option with `--emit=code`, `--text -c`, or `--disassemble`.

Use this option with `--source_directory` if you want to specify additional paths to search for source code.

Related information

--disassemble on page 789
--emit=option[,option,...] on page 791
--source_directory=path on page 823
--text on page 826

4.1.43 --linkview, --no_linkview

Controls the section-level view from the ELF image.



Note

This option is deprecated.

Syntax

```
--linkview
--no_linkview
```

Parameters

None.

Restrictions

The following restrictions apply:

You must use `--elf` with `--linkview` and `--no_linkview`.

Operation

`--no_linkview` discards the section-level view and retains only the segment-level view (load time view).

Discarding the section-level view eliminates:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The section header table.
- The section header string table.
- The string table.
- The symbol table.
- All debug sections.

All that is left in the output is the program header table and the program segments.

Example

To get ELF format output for image.axf, enter:

```
fromelf --no_linkview --elf image.axf --output=image_nlk.axf
```

Related information

--elf on page 791
--privacy (fromelf) on page 816
--strip=option[,option,...] on page 823
--privacy (armlink) on page 520

4.1.44 --localize=option[,option,...]

Converts the selected symbols to local symbols.

Syntax

```
--localize=<option>[,<option>,...]
```

Parameters

<option>

One of:

<object_name>::

All symbols in ELF objects with a name matching <object_name> are converted to local symbols.

<object_name>::<symbol_name>

All symbols in ELF objects with a name matching <object_name> and also a symbol name matching <symbol_name> are converted to local symbols.

<symbol_name>

All symbols with a symbol name matching <symbol_name> are converted to local symbols.

You can:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Use wildcard characters `?` and `*` for symbolic names in `<symbol_name>` and `<object_name>` arguments
- Specify multiple values in one `<option>` followed by a comma-separated list of arguments.

Restrictions

You must use `--elf` with this option.

Related information

--elf on page 791
--hide=option[,option,...] on page 801

4.1.45 --m32

Produces Motorola 32-bit format (32-bit S-records) output. It generates one output file for each load region in the image.

You can specify the base address of the output with the `--base` option.

Syntax

```
--m32
```

Parameters

None.

Restrictions

The following restrictions apply:

- Not supported for AArch64 state.
- You cannot use this option with object files.
- You must use `--output` with this option.

Operation

Considerations when using `--m32`

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named `<destination>` and generates one binary output file for each load region in the input image. `fromelf` places the output files in the `<destination>` directory.


Note

For multiple load regions, the name of the first non-empty execution region in the corresponding load region is used for the filename.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A file is only created when the load region describes code or data that is present in the ELF file. For example a load region containing only execution regions with ZI data in them does not result in an output file.

Example

To convert the ELF file infile.axf to a Motorola 32-bit format file, for example `outfile.bin`, enter:

```
fromelf --m32 --output=outfile.bin infile.axf
```

Related information

--base [[object_file::]load_region_ID=]num on page 767
--m32combined on page 813
--output=destination on page 815

4.1.46 --m32combined

Produces Motorola 32-bit format (32-bit S-records) output. It generates one output file for an image containing multiple load regions.

You can specify the base address of the output with the `--base` option.

Syntax

```
--m32combined
```

Parameters

None.

Restrictions

The following restrictions apply:

- Not supported for AArch64 state.
- You cannot use this option with object files.
- You must use `--output` with this option.

Operation

Considerations when using `--m32combined`

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named `<destination>` and generates one binary output file for all load regions in the input image. `fromelf` places the output file in the `<destination>` directory.

ELF images contain multiple load regions if, for example, they are built with a scatter file that defines more than one load region.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

To create a single Motorola 32-bit format output file, `outfile2.bin`, from an image file `infile2.axf`, with two load regions, and with a start address of `0x1000`, enter:

```
fromelf --m32combined --base=0x1000 --output=outfile2.bin infile2.axf
```

Related information

--base [[object_file::]load_region_ID=]num on page 767
--m32 on page 812
--output=destination on page 815

4.1.47 --only=section_name

Filters the list of sections that are displayed in the main section-by-section output from `--text`. It does not affect any additional output after the main section-by-section output.

Syntax

```
--only=<section_name>
```

Parameters

`<section_name>`

The name of the section to be displayed.

You can:

- Use wildcard characters `?` and `*` for a section name.
- Use multiple `--only` options to specify additional sections to display.

Example: Display only the symbol table

To display only the symbol table, `.symtab`, from the section-by-section output, enter:

```
fromelf --only=.symtab --text -s test.axf
```

Example: Display all ER<n> sections

To display all `ER<n>` sections, enter:

```
fromelf --only=ER? test.axf
```

Example: Display the HEAP section and all symbol and string table sections

To display the `HEAP` section and all symbol and string table sections, enter:

```
fromelf --only=HEAP --only=.*tab --text -s -t test.axf
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--text on page 826

4.1.48 --output=destination

Specifies the name of the output file, or the name of the output directory if multiple output files are created.

Syntax

```
--output=<destination>
-o <destination>
```

Parameters

<destination>

Either a file or a directory. For example:

--output=foo

The name of an output file

--output=foo/

The name of an output directory.

Operation

With `--bin` or `--elf`:

- You can specify a single input file and a single output filename.
- If you specify many input files and use `--elf`, you can use `--in_place` to write the output of processing each file over the top of the input file.
- If you specify many input filenames and specify an output directory, then the output from processing each file is written into the output directory. Each output filename is derived from the corresponding input file. Therefore, specifying an output directory in this way is the only method of converting many ELF files to a binary or hexadecimal format in a single run of `fromelf`.
- If you specify an archive file as the input, then the output file is also an archive. For example, the following command creates an archive file called `output.o`:

```
fromelf --elf --strip=debug archive.a --output=output.o
```

- If you specify a pattern in parentheses to select a subset of objects from an archive, fromelf only converts the subset. All the other objects are passed through to the output archive unchanged.

Related information

--bin on page 768
--elf on page 791
--text on page 826

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.49 --privacy (fromelf)

Modifies the output file to protect your code in images and objects that are delivered to third parties.

Syntax

```
--privacy
```

Parameters

None.

Operation

The effect of this option is different for images and object files.

For images, this option:

- Changes section names to a default value, for example, changes code section names to `.text`
- Removes the complete symbol table in the same way as `--strip symbols`
- Removes the `.comment` section name, and is marked as `[Anonymous Section]` in the `fromelf --text` output.

For object files, this option:

- Changes section names to a default value, for example, changes code section names to `.text`.
- Keeps mapping symbols and build attributes in the symbol table.
- Removes those local symbols that can be removed without loss of functionality.

  Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.

Related information

--strip=option[,option,...] on page 823
--locals, --no_locals on page 497
--privacy (armlink) on page 520

4.1.50 --qualify

Modifies the effect of the `--fieldoffsets` option so that the name of each output symbol includes an indication of the source file containing the relevant structure.

Syntax

```
--qualify
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

This enables the `--fieldoffsets` option to produce functional output even if two source files define different structures with the same name.

If the source file is in a different location from the current location, then the source file path is also included.

A structure called `test` is defined in two headers, for example, `one.h` and `two.h`.

Using `fromelf` option `--fieldoffsets`, the linker might define the following symbols:

- `test.a`, `test.b`, and `test.c`.
- `test.x`, `test.y`, and `test.z`.

Using `fromelf` options `--qualify --fieldoffsets`, the linker defines the following symbols:

- `oneh_test.a`, `oneh_test.b` and `oneh_test.c`.
- `twoh_test.x`, `twoh_test.y` and `twoh_test.z`.

Related information

--fieldoffsets on page 796

4.1.51 --relax_section=option[,option,...]

Changes the severity of a compare report for the specified sections to warnings rather than errors.

Syntax

```
--relax_section=<option>[,<option>,...]
```

Parameters

`<option>`

One of:

`<object_name>::`

All sections in ELF objects with a name matching `<object_name>`.

`<object_name>::<section_name>`

All sections in ELF objects with a name matching `<object_name>` and also a section name matching `<section_name>`.

`<section_name>`

All sections with a name matching `<section_name>`.

You can:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Use wildcard characters `?` and `*` for symbolic names in `<section_name>` and `<object_name>` arguments
- Specify multiple values in one `<option>` followed by a comma-separated list of arguments.

Restrictions

You must use `--compare` with this option.

Related information

--compare=option[,option,...] on page 775
--ignore_section=option[,option,...] on page 804
--relax_symbol=option[,option,...] on page 818

4.1.52 --relax_symbol=option[,option,...]

Changes the severity of a compare report for the specified symbols to warnings rather than errors.

Syntax

```
--relax_symbol=<option>[,<option>,...]
```

Parameters

`<option>`

One of:

`<object_name>::`

All symbols in ELF objects with a name matching `<object_name>`.

`<object_name>::<symbol_name>`

All symbols in ELF objects with a name matching `<object_name>` and also a symbol name matching `<symbol_name>`.

`<symbol_name>`

All symbols with a name matching `<symbol_name>`.

You can:

- Use wildcard characters `?` and `*` for symbolic names in `<symbol_name>` and `<object_name>` arguments
- Specify multiple values in one `<option>` followed by a comma-separated list of arguments.

Restrictions

You must use `--compare` with this option.

Related information

--compare=option[,option,...] on page 775

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--ignore_symbol=option[,option,...] on page 805
--relax_section=option[,option,...] on page 817

4.1.53 --rename=option[,option,...]

Renames the specified symbol in an output ELF object.

Syntax

```
--rename=<option>[,<option>,...]
```

Parameters

<option>

One of:

<object_name>::<old_symbol_name=new_symbol_name>

This replaces all symbols in the ELF object <object_name> that have a symbol name matching <old_symbol_name>.

<old_symbol_name=new_symbol_name>

This replaces all symbols that have a symbol name matching <old_symbol_name>.

You can:

- Use wildcard characters ? and * for symbolic names in <old_symbol_name>, <new_symbol_name>, and <object_name> arguments.
- Specify multiple values in one <option> followed by a comma-separated list of arguments.

Restrictions

You must use --elf and --output with this option.

Example: Rename the clock symbol in the timer.axf image to myclock

To rename the clock symbol in the timer.axf image to myclock, and create a new file called mytimer.axf:

```
fromelf --elf --rename=clock=myclock --output=mytimer.axf timer.axf
```

Related information

--elf on page 791
--output=destination on page 815

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.54 --select=select_options

When used with `--fieldoffsets` or `--text -a` options, displays only those fields that match a specified pattern list.

Syntax

```
--select=<select_options>
```

Parameters

`<select_options>`

A list of patterns to match. Use special characters to select multiple fields:

- Use a comma-separated list to specify multiple fields, for example:

  ```
  a*,b*,c*
  ```

- Use the wildcard character `*` to match any name.
- Use the wildcard character `?` to match any single letter.
- Prefix the `<select_options>` string with `+` to specify the fields to include. This is the default behavior.
- Prefix the `<select_options>` string with `~` to specify the fields to exclude.

If you are using a special character on Unix platforms, you must enclose the options in quotes to prevent the shell expanding the selection.

Operation

Use this option with either `--fieldoffsets` or `--text -a`.

Example: Output from the -fieldoffsets option

The output from the `--fieldoffsets` option might include the following data structure:

```
|structure.f1|    EQU     0         ;  int16_t
|structure.f2|    EQU     0x2       ;  int16_t
|structure.f3|    EQU     0x4       ;  int16_t
|structure.f11|   EQU     0x6       ;  int16_t
|structure.f21|   EQU     0x8       ;  int16_t
|structure.f31|   EQU     0xA       ;  int16_t
|structure.f111|  EQU     0xC       ;  int16_t
```

To output only those fields that start with `f1`, enter:

```
fromelf --select=structure.f1* --fieldoffsets infile.axf
```

This command produces the output:

```
|structure.f1|    EQU     0         ;  int16_t
|structure.f11|   EQU     0x6       ;  int16_t
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
|structure.f111|  EQU     0xC      ;  int16_t
                  END
```

Related information

--fieldoffsets on page 796
--text on page 826

4.1.55 --show=option[,option,...]

Changes the symbol visibility property of the selected symbols, to mark them with default visibility.

Syntax

```
--show=<option>[,<option>,...]
```

Parameters

<option>

One of:

<object_name>::

All symbols in ELF objects with a name matching <object_name> are marked as having default visibility.

<object_name>::<symbol_name>

All symbols in ELF objects with a name matching <object_name> and also a symbol name matching <symbol_name> are marked as having default visibility.

<symbol_name>

All symbols with a symbol name matching <symbol_name> are marked as having default visibility.

You can:

- Use wildcard characters ? and * for symbolic names in <symbol_name> and <object_name> arguments
- Specify multiple values in one <option> followed by a comma-separated list of arguments.

Restrictions

You must use --elf with this option.

Related information

--elf on page 791
--hide=option[,option,...] on page 801

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.56 --show_and_globalize=option[,option,...]

Changes the symbol visibility property of the selected symbols, to mark them with default visibility, and converts the selected symbols to global symbols.

Syntax

```
--show_and_globalize=<option>[,<option>,...]
```

Parameters

<option>

One of:

<object_name>::

All symbols in ELF objects with a name matching `<object_name>`.

<object_name>::<symbol_name>

All symbols in ELF objects with a name matching `<object_name>` and also a symbol name matching `<symbol_name>`.

<symbol_name>

All symbols with a symbol name matching `<symbol_name>`.

You can:

- Use wildcard characters `?` and `*` for symbolic names in `<symbol_name>` and `<object_name>` arguments
- Specify multiple values in one `<option>` followed by a comma-separated list of arguments.

Restrictions

You must use `--elf` with this option.

Related information

--elf on page 791

4.1.57 --show_cmdline (fromelf)

Outputs the `fromelf` command line.

Syntax

```
--show_cmdline
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

Shows the command line after processing by the tool, and can be useful to check:

- The command line a build system is using.
- How the tool 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 information

--via=file (fromelf) on page 831

4.1.58 --source_directory=path

Explicitly specifies the location of the source code.

Syntax

```
--source_directory=<path>
```

Parameters

`<path>`

The location of the source code.

Operation

By default, the source code is assumed to be located in a directory relative to the ELF input file. You can use this option multiple times to specify a search path involving multiple directories.

You can use this option with `--interleave`.

Related information

--interleave=option on page 809

4.1.59 --strip=option[,option,...]

Helps to protect your code in images and objects that are delivered to third parties. You can also use it to help reduce the size of the output image.

Syntax

```
--strip=<option>[,<option>,...]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<option>

One of:

all

For object modules, this option removes all debug, comments, notes and symbols from the ELF file. For executables, this option works the same as `--no_linkview`.

debug

Removes all debug sections from the ELF file.

comment

Removes the .comment section from the ELF file.

filesymbols

The `STT_FILE` symbols are removed from the ELF file.

localsymbols

The effect of this option is different for images and object files.

For images, this option removes all local symbols, including mapping symbols, from the output symbol table.

For object files, this option:

- Keeps mapping symbols and build attributes in the symbol table.
- Removes those local symbols that can be removed without loss of functionality.

  Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.

notes

Removes the .notes section from the ELF file.

pathnames

Removes the path information from all symbols with type `STT_FILE`. For example, an `STT_FILE` symbol with the name `C:\work\myobject.o` is renamed to `myobject.o`.


Note

This option does not strip path names that are in the debug information.

symbols

The effect of this option is different for images and object files.

For images, this option removes the complete symbol table, and all static symbols. If any of these static symbols are used as a static relocation target, then these relocations are also removed. In all cases, `STT_FILE` symbols are removed.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

For object files, this option:

- Keeps mapping symbols and build attributes in the symbol table.
- Removes those local symbols that can be removed without loss of functionality.

  Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.


Note

Stripping the symbols, path names, or file symbols might make the file more difficult to debug.

Restrictions

You must use `--elf` and `--output` with this option.

Example: Produce an output file without debug from an input ELF file

To produce an `output.axf` file without debug from the ELF file `infile.axf` originally produced with debug, enter:

```
fromelf --strip=debug,symbols --elf --output=outfile.axf infile.axf
```

Related information

--elf on page 791
--linkview, --no_linkview on page 810
--privacy (fromelf) on page 816
About mapping symbols on page 622
--locals, --no_locals on page 497
--privacy (armlink) on page 520

4.1.60 --symbolversions, --no_symbolversions

Turns off the decoding of symbol version tables.

Syntax

```
--symbolversions
--no_symbolversions
```

Parameters

None.

Restrictions

If you use `--elf` with this option, you must also use `--output`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Symbol versioning on page 756

4.1.61 --text

Prints image information in text format. You can decode an ELF image or ELF object file using this option.

Syntax

```
--text [<options>]
```

Parameters

<options>

Specifies what is displayed, and can be one or more of the following:

-a

Prints the global and static data addresses, including addresses for structure and union contents.

This option can only be used on files containing debug information. If no debug information is present, a warning is displayed.

Use the `--select` option to output a subset of fields in a data structure.

If you want to view the data addresses of arrays, expanded both inside and outside structures, use the `--expandarrays` option with this text category.

-c

This option disassembles code, alongside a dump of the original binary data being disassembled and the addresses of the instructions.


Note

Disassembly is generated in `armasm` assembler syntax and not GNU assembler syntax.

Unlike `--disassemble`, the disassembly cannot be used as input to `armasm`.


Note

To disassemble *M-profile Vector Extension* (MVE) instructions, you must specify the `--cpu` option for Arm®v8.1-M and later architectures. For example, `--cpu =8.1-M.Main.mve`.

To disassemble *Scalable Vector Extension* (SVE) instructions, you must not specify the `--cpu` option. `fromelf` cannot disassemble Armv8.4-A and later instructions without also disassembling SVE instructions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

`armasm` does not support:

- Armv8.4-A and later architectures.
- Certain backported options in Armv8.2-A and Armv8.3-A.
- Assembling MVE or SVE instructions.

`-d`

Prints contents of the data sections.

`-e`

Decodes exception table information for objects. Use with `-c` when disassembling images.


Note

Not supported for AArch64 state.

`-g`

Prints debug information.

`-r`

Prints relocation information.

`-s`

Prints the symbol and versioning tables.

`-t`

Prints the string tables.

`-v`

Prints detailed information on each segment and section header of the image.

`-w`

Eliminates line wrapping.

`-y`

Prints dynamic segment contents.

`-z`

Prints the code and data sizes.

These options are only recognized in text mode. In text mode, you do not have to specify `--text`.

Restrictions

`-e` is not supported for AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

If you do not specify a code output format, `--text` is assumed. That is, you can specify one or more options without having to specify `--text`. For example, `fromelf -a` is the same as `fromelf --text -a`.

If you specify a code output format, such as `--bin` , then any `--text` options are ignored.

If `<destination>` is not specified with the `--output` option, or `--output` is not specified, the information is displayed on `stdout`.

Use the `--only` option to filter the list of sections.

Example: Produce a plain text output file that contains the disassembly

To produce a plain text output file that contains the disassembled version of an ELF image and the symbol table, enter:

```
fromelf --text -c -s --output=outfile.lst infile.axf
```

Example: List all the global and static data variables and all the structure field addresses to stdout

To list to `stdout` all the global and static data variables and all the structure field addresses, enter:

```
fromelf -a --select=* infile.axf
```

Example: Produce a text file containing all of the structure addresses but none of the global or static data variable information

To produce a text file containing all of the structure addresses in `infile.axf` but none of the global or static data variable information, enter:

```
fromelf --text -a --select=*.* --output=structaddress.txt infile.axf
```

Example: Produce a text file containing addresses of the nested structures only

To produce a text file containing addresses of the nested structures only, enter:

```
fromelf --text -a --select=*.*.* --output=structaddress.txt infile.axf
```

Example: Produce a text file containing all of the global or static data variable information but none of the structure addresses

To produce a text file containing all of the global or static data variable information in `infile.axf` but none of the structure addresses, enter:

```
fromelf --text -a --select=*,~*.* --output=structaddress.txt infile.axf
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Output only the .symtab section information

To output only the `.symtab` section information in `infile.axf`, enter:

```
fromelf --only .symtab -s --output=symtab.txt infile.axf
```

Related information

--cpu=name (fromelf) on page 779
--emit=option[,option,...] on page 791
--expandarrays on page 793
--info=topic[,topic,...] (fromelf) on page 806
--interleave=option on page 809
--only=section_name on page 814
--output=destination on page 815
--select=select_options on page 819
-w on page 832
--disassemble on page 789
Using fromelf to find where a symbol is placed in an executable ELF image
Getting Image Details

4.1.62 --version_number (fromelf)

Displays the version of fromelf that you are using.

Syntax

```
--version_number
```

Parameters

None.

Operation

`fromelf` displays the version number in the format `<Mmmuuxx>`, where:

- `<M>` is the major version number, `6`.
- `<mm>` is the minor version number.
- `<uu>` is the update number.
- `<xx>` is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions.

Related information

--help (fromelf) on page 800
--vsn (fromelf) on page 831

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.63 --vhx

Produces Byte oriented (Verilog Memory Model) hexadecimal format output.

Syntax

```
--vhx
```

Parameters

None.

Restrictions

The following restrictions apply:

- You cannot use this option with object files.
- You must use `--output` with this option.

Operation

This format is suitable for loading into the memory models of Hardware Description Language (HDL) simulators. You can split output from this option into multiple files with the `--<width>x<banks>` option.

Considerations when using `--vhx`

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named `<destination>` and generates one binary output file for each load region in the input image. `fromelf` places the output files in the `<destination>` directory.


Note

For multiple load regions, the name of the first non-empty execution region in the corresponding load region is used for the filename.

A file is only created when the load region describes code or data that is present in the ELF file. For example a load region containing only execution regions with ZI data in them does not result in an output file.

Example: Convert an ELF file to a byte oriented hexadecimal format file

To convert the ELF file `infile.axf` to a byte oriented hexadecimal format file, for example `outfile.bin`, enter:

```
fromelf --vhx --output=outfile.bin infile.axf
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example: Create multiple output files in the regions directory with two 8-bit memory banks

To create multiple output files, in the regions directory, from an image file `multiload.axf`, with two 8-bit memory banks, enter:

```
fromelf --vhx --8x2 multiload.axf --output=regions
```

Related information

--output=destination on page 815
--widthxbanks on page 834

4.1.64 --via=file (fromelf)

Reads an additional list of input filenames and tool options from `<filename>`.

Syntax

```
--via=<filename>
```

Parameters

`<filename>`

The name of a via file containing options to be included on the command line.

Operation

You can enter multiple `--via` options on the `armasm`, `armlink`, `fromelf`, and `armar` command lines. You can also include the `--via` options within a via file.

Related information

Via File Syntax on page 1082
Via file syntax rules on page 1083

4.1.65 --vsn (fromelf)

Displays the version information and the license details.

Syntax

```
--vsn
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm® Compiler for Embedded tool you are using. If you need to access the version in other tools or scripts, for example in build scripts, use the output from `--version_number`.

Table 4-2: Arm Compiler for Embedded Version Information

Field	Description
`Product: <Product Name>`	The product name, which depends on the currently activated license or suite that contains Arm Compiler for Embedded as a component, such as: • Arm *Software Success Kit* (SSK) with an SSK license entitlement. • Arm Development Studio. • Arm® Keil® *Microprocessor Development Kit* (MDK). For example, in Arm Development Studio, `--vsn` lists the product as: `Arm Development Studio <edition> Edition <version>`
`Component: <component>`	The version of Arm Compiler for Embedded. The version is made up of the major, minor, and update number: `Arm Compiler for Embedded <Major Version>.<Minor Version>.<Update Version>`
`Tool: <tool name> [<tool identifier>]`	The tool name and tool identifier. The tool identifier must not be used to identify versions of Arm Compiler for Embedded. The tool identifier is different for each operating system.

Example: fromelf --vsn output for Arm Development Studio

The `fromelf` output for Arm Compiler for Embedded 6.23 on Linux x86 with Arm Development Studio Gold Edition 2024.0:

```
Product: Arm Development Studio Gold Edition 2024.0
Component: Arm Compiler for Embedded 6.23
Tool: fromelf [<toolid>]
```

Related information

--help (fromelf) on page 800
--version_number (fromelf) on page 829

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.66 -w

Causes some text output information that usually appears on multiple lines to be displayed on a single line.

Syntax

```
-w
```

Parameters

None.

Operation

This option makes the output easier to parse with text processing utilities such as Perl.

Example

```
>  fromelf --text -w -c test.axf

============================================================================
** ELF Header Information
.
.
.
============================================================================
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]    Size   : 36
 bytes (alignment 4)    Address: 0x00000000    $a
     .text
.
.
.
** Section #7 '.rel.text' (SHT_REL)    Size   : 8 bytes (alignment 4)    Symbol
 table #6 '.symtab'    1 relocations applied to section #1 '.text'
** Section #2 '.ARM.exidx' (SHT_ARM_EXIDX) [SHF_ALLOC + SHF_LINK_ORDER]    Size   :
 8 bytes (alignment 4)    Address: 0x
00000000    Link to section #1 '.text'
** Section #8 '.rel.ARM.exidx' (SHT_REL)    Size   : 8 bytes (alignment 4)    Symbol
 table #6 '.symtab'    1 relocations applied to section #2 '.ARM.exidx'
** Section #3 '.arm_vfe_header' (SHT_PROGBITS)    Size   : 4 bytes (alignment 4)
** Section #4 '.comment' (SHT_PROGBITS)    Size   : 74 bytes
** Section #5 '.debug_frame' (SHT_PROGBITS)    Size   : 140 bytes
** Section #9 '.rel.debug_frame' (SHT_REL)    Size   : 32 bytes (alignment 4)
 Symbol table #6 '.symtab'    4 relocations applied to section #5 '.debug_frame'
** Section #6 '.symtab' (SHT_SYMTAB)    Size   : 176 bytes (alignment 4)    String
 table #11 '.strtab'    Last local symbol no. 5
** Section #10 '.shstrtab' (SHT_STRTAB)    Size   : 110 bytes
** Section #11 '.strtab' (SHT_STRTAB)    Size   : 223 bytes
** Section #12 '.ARM.attributes' (SHT_ARM_ATTRIBUTES)    Size   : 69 bytes
```

Related information

--text on page 826

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

4.1.67 --wide64bit

Causes all addresses to be displayed with a width of 64 bits.

Syntax

```
--wide64bit
```

Parameters

None.

Operation

Without this option fromelf displays addresses as 32 bits where possible, and only displays them as 64 bits when necessary.

This option is ignored if the input file is not an AArch64 state file.

Related information

input_file (fromelf) on page 808

4.1.68 --widthxbanks

Outputs multiple files for multiple memory banks.

Syntax

```
--<width>x<banks>
```

Parameters

<banks>

specifies the number of memory banks in the target memory system. It determines the number of output files that are generated for each load region.

<width>

is the width of memory in the target memory system (8-bit, 16-bit, 32-bit, or 64-bit).

Valid configurations for <banks>x<width>

```
--8x1
--8x2
--8x4
--16x1
--16x2
--32x1
--32x2
--64x1
```

Restrictions

You must use `--output` with this option.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

`fromelf` uses the last specified configuration if more than one configuration is specified.

If the image has one load region, `fromelf` generates the same number of files as the number of `<banks>` specified. The filenames are derived from the `--output=<destination>` argument, using the following naming conventions:

- If there is one memory bank (`<banks>` = 1) the output file is named `<destination>`.
- If there are multiple memory banks (`<banks>` > 1), `fromelf` generates `<banks>` number of files named `<destinationN>` where `<N>` is in the range `0` to `<banks>-1`. If you specify a file extension for the output filename, then the number `<N>` is placed before the file extension. For example:

  ```
  fromelf --cpu=8-A.32 --vhx --8x2 test.axf --output=test.txt
  ```

  This generates two files named `test0.txt` and `test1.txt`.

If the image has multiple load regions, `fromelf` creates a directory named `<destination>` and generates `<banks>` files for each load region in that directory. The files for each load region are named `<load_regionN>` where `<load_region>` is the name of the load region, and `<N>` is in the range `0` to `<banks>-1`. For example:

```
fromelf --cpu=8-A.32 --vhx --8x2 multiload.axf --output=regions/
```

This might produce the following files in the `regions` directory:

```
EXEC_ROM0
EXEC_ROM1
RAM0
RAM1
```

The memory width specified by `<width>` controls the amount of memory that is stored in a single line of each output file. The size of each output file is the size of memory to be read divided by the number of files created. For example:

- `fromelf --cpu=8-A.32 --vhx --8x4 test.axf --output=file` produces four files (`file0`, `file1`, `file2`, and `file3`). Each file contains lines of single bytes, for example:

  ```
  00
  00
  2D
  00
  2C
  8F
  ...
  ```

- `fromelf --vhx --16x2 test.axf --output=file` produces two files (`file0` and `file1`). Each file contains lines of two bytes, for example:

  ```
  0000
  002D
  002C
  ...
  ```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--bin on page 768
--output=destination on page 815
--vhx on page 829

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5. armar Reference

A list of the command-line options for the `armar` command, and other reference information that is relevant to `armar`.

5.1 armar Command-line Options

Describes the command-line options of the Arm librarian, `armar`.

5.1.1 archive

Specifies the location of the library to be created, modified, or read.


Note

If you include a list of files in `<file_list>`, they must be specified after the library file.

Related information

file_list on page 845

5.1.2 -a pos_name

Places new files in the library after the specified library member.

Syntax

```
-a=<pos_name>
```

Parameters

`<pos_name>`

The name of a file in the library.

Operation

The effect of this option is negated if you include `-b` or `-i` on the same command line.

Example

To add or replace files `obj3.o` and `obj4.o` immediately after `obj2.o` in `mylib.a`, enter:

```
armar -r -a obj2.o mylib.a obj3.o obj4.o
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

-b pos_name on page 838
-i pos_name on page 845
-m pos_name (armar) on page 846
-r on page 848

5.1.3 -b pos_name

Places new files in the library before the specified library member.

Syntax

```
-b=<pos_name>
```

Parameters

<pos_name>
The name of a file in the library.

Operation

This option takes precedence if you include `-a` on the same command line.

Related information

-a pos_name on page 837
-i pos_name on page 845
-m pos_name (armar) on page 846
-r on page 848

5.1.4 -c (armar)

Suppresses the diagnostic message normally written to `stderr` when a library is created.

Syntax

```
-c
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.1.5 -C (armar)

Instructs the librarian not to replace existing files with like-named files when performing extractions.

Syntax

```
-C
```

Parameters

None.

Operation

Use this option with `-T` to prevent truncated filenames from replacing files with the same prefix.

An error message is displayed if the file to be extracted already exists in the current location.

Related information

-T on page 852
-x (armar) on page 856

5.1.6 --create

Creates a new library containing only the files specified in `<file_list>`. If the library already exists, its previous contents are discarded.

Syntax

```
--create
```

Parameters

None.

Operation

With the `--create` option specify the list of object files, either:

- Directly on the command-line.
- In a via file.

You can use this option together with the following compatible command-line options:

- `-c`
- `--diag_style`
- `-n`
- `-v`
- `--via`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

Other options can also create a new library in some circumstances. For example, using the `-r` option with a library that does not exist.

Examples

To create a new library by adding all object files in the current directory, enter:

```
armar --create mylib.a *.o
```

To create a new library containing the files listed in a via file, enter:

```
armar --create mylib.a --via myobject.via
```

Related information

file_list on page 845

5.1.7 -d (armar)

Deletes one or more files specified in `<file_list>` from the library.

Syntax

```
-d
```

Parameters

None.

Example

To delete the files `file1.o` and `file2.o` from the `mylib.a` library, enter:

```
armar -d mylib.a file1.o file2.o
```

Related information

file_list on page 845

5.1.8 --debug_symbols

By default, debug symbols are excluded from an archive. Use `--debug_symbols` to include debug symbols in the archive.

Syntax

```
--debug_symbols
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Related information

About the Librarian

5.1.9 --diag_error=tag[,tag,...] (armar)

Sets diagnostic messages that have a specific tag to Error severity.

Syntax

```
--diag_error={<tag>[,<tag>,...]}
```

Parameters

<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.

Related information

--diag_remark=tag[,tag,...] (armar) on page 841
--diag_style=arm|ide|gnu (armar) on page 842
--diag_suppress=tag[,tag,...] (armar) on page 842
--diag_warning=tag[,tag,...] (armar) on page 843

5.1.10 --diag_remark=tag[,tag,...] (armar)

Sets diagnostic messages that have a specific tag to Remark severity.

Syntax

```
--diag_remark={<tag>[,<tag>,...]}
```

Parameters

<tag>

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 information

--diag_error=tag[,tag,...] (armar) on page 841
--diag_style=arm|ide|gnu (armar) on page 842

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--diag_suppress=tag[,tag,...] (armar) on page 842
--diag_warning=tag[,tag,...] (armar) on page 843

5.1.11 --diag_style=arm|ide|gnu (armar)

Specifies the display style for diagnostic messages.

Default

The default is `--diag_style=arm`.

Syntax

```
--diag_style=<string>
```

Parameters

`<string>`

One of:

`arm`

Display messages using the legacy 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`.

Operation

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

Related information

--diag_error=tag[,tag,...] (armar) on page 841
--diag_remark=tag[,tag,...] (armar) on page 841
--diag_suppress=tag[,tag,...] (armar) on page 842
--diag_warning=tag[,tag,...] (armar) on page 843

5.1.12 --diag_suppress=tag[,tag,...] (armar)

Suppresses diagnostic messages that have a specific tag.

Syntax

```
--diag_suppress={<tag>[,<tag>,...]}
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<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 information

--diag_error=tag[,tag,...] (armar) on page 841
--diag_remark=tag[,tag,...] (armar) on page 841
--diag_style=arm|ide|gnu (armar) on page 842
--diag_warning=tag[,tag,...] (armar) on page 843

5.1.13 --diag_warning=tag[,tag,...] (armar)

Sets diagnostic messages that have a specific tag to Warning severity.

Syntax

```
--diag_warning={<tag>[,<tag>,...]}
```

Parameters

<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.

Related information

--diag_error=tag[,tag,...] (armar) on page 841
--diag_remark=tag[,tag,...] (armar) on page 841
--diag_style=arm|ide|gnu (armar) on page 842
--diag_suppress=tag[,tag,...] (armar) on page 842

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.1.14 --entries

Lists all object files in the library that have an entry point. You can use the `armasm` directive `ENTRY` to specify an entry point in legacy `armasm` syntax assembler code.

Syntax

```
--entries
```

Parameters

None.

Operation

For objects created from `armasm` legacy assembler containing the `ENTRY` directive, the format for the listing is:

```
ENTRY at offset <num> in section <name> of <member>
```


Note

This option outputs the `No ENTRY points found.` message in the following cases:

- If a library contains only objects created from GNU syntax assembler.
- If a library contains only objects created from `armasm` legacy assembler without the `ENTRY` directive.
- If a library contains a mixture of these objects.

Example

The following example lists the entry point of each `armasm` legacy assembler object file in `myasm.a` containing an `ENTRY` directive:

```
> armar --entries myasm.a

ENTRY at offset 0 in section adrlabel of adrlabel.o
ENTRY at offset 0 in section ARMex of armex.o
ENTRY at offset 0 in section Block of blocks.o
ENTRY at offset 0 in section Jump of jump.o
ENTRY at offset 0 in section LDRlabel of ldrlabel.o
ENTRY at offset 0 in section Loadcon of loadcon.o
ENTRY at offset 0 in section StrCopy of strcopy.o
ENTRY at offset 0 in section subrout of subrout.o
ENTRY at offset 0 in section Tblock of tblock.o
ENTRY at offset 0 in section ThumbSub of thumbsub.o
ENTRY at offset 0 in section Word of word.o
```

Related information

--sizes on page 850
--zt on page 857
ENTRY directive on page 991
Miscellaneous directives

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.1.15 file_list

A space-separated list of ELF-compliant files, such as ELF objects and ELF libraries.

Operation

Each file must be fully specified by its path and name. The path can be absolute, relative to drive and root, or relative to the current directory.


Note

The list of files must be specified after the library file.

Only the filename at the end of the path is used when comparing against the names of files in the library. If two or more path operands end with the same filename, the results are unspecified. You can use the wild characters `*` and `?` to specify files.

If one of the files is a library, `armar` copies all members from the input library to the destination library. The order of members on the command line is preserved. Therefore, supplying a library file is logically equivalent to supplying all of its members in the order that they are stored in the library.

5.1.16 --help (armar)

Displays a summary of the main command-line options.

Default

This option is the default if you specify the tool command without any options or source files.

Syntax

```
--help
```

Parameters

None.

Related information

--version_number (armar) on page 853
--vsn (armar) on page 854

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.1.17 -i pos_name

Places new files in the library before the specified library member.

Syntax

```
-i <pos_name>
```

Parameters

<pos_name>

The name of a file in the library.

This option is equivalent to `-b <pos_name>`.

Related information

-a pos_name on page 837
-b pos_name on page 838
-m pos_name (armar) on page 846
-r on page 848

5.1.18 -m pos_name (armar)

Moves files in a library to a specified position.

Syntax

```
-m=<pos_name>
```

Parameters

<pos_name>

The name of a file in the library.

Operation

If `-a`, `-b`, or `-i` with `<pos_name>` is specified, moves files to the new position. Otherwise, moves files to the end of the library.

Example

To move the file `file1.o` to a new location after `file2.o` in the `mylib.a` library, enter:

```
armar -m -a file2.o mylib.a file1.o
```

Related information

-a pos_name on page 837
-b pos_name on page 838
-i pos_name on page 845

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.1.19 -n

Suppresses the creation of a symbol table in the library.

Syntax

```
-n
```

Parameters

None.

Operation

By default, `armar` always creates a symbol table when you create a library of object files.

You can recreate the symbol table in the library using the `-s` option.

Example

To create a library without a symbol table, enter:

```
armar -n --create mylib.a *.obj
```

Related information

-s (armar) on page 849

5.1.20 --new_files_only

Updates an object file in the archive only if the new object has a later timestamp.

Syntax

```
--new_files_only
```

Parameters

None.

Operation

When used with the `-r` option, files in the library are replaced only if the corresponding file has a modification time that is newer than the modification time of the file in the library.

Related information

-r on page 848
-u (armar) on page 852

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.1.21 -p

Prints the contents of source files in a library to `stdout`.


Note

The files must be text files.

Syntax

```
-p
```

Parameters

None.

Example

To display the contents of `file1.c` in `mylib.a`, enter:

```
armar -p mylib.a file1.c
```

Related information

-t on page 851

5.1.22 -r

Replaces, or adds, files in the specified library.

Syntax

```
-r
```

Parameters

None.

Operation

If the library does not exist, a new library file is created and a diagnostic message is written to standard error. You can use this option in conjunction with other compatible command-line options.

`-q` is an alias for `-r`.

If no files are specified and the library exists, the results are undefined. Files that replace existing files do not change the order of the library.

If the `-u` option is used, then only those files with dates of modification later than the library files are replaced.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If the `-a`, `-b`, or `-i` option is used, then `<pos_name>` must be present and specifies that new files are to be placed after (`-a`) or before (`-b` or `-i`) `<pos_name>`. Otherwise the new files are placed at the end.

Examples

To add or replace `obj1.o`, `obj2.o`, and `obj3.o` files in a library, enter:

```
armar -r mylib.a obj1.o obj2.o obj3.o
```

To replace files with names beginning with `k` in a library, and only if the file in the library is older than the specified file, enter:

```
armar -ru mylib.a k*.o
```

Related information

-a pos_name on page 837
-b pos_name on page 838
-i pos_name on page 845
-u (armar) on page 852
file_list on page 845

5.1.23 -s (armar)

Creates a symbol table in the library.

Syntax

```
-s
```

Parameters

None.

Operation

This option is useful for libraries that have been created:

- Using the `-n` option.
- With an archiver that does not automatically create a symbol table.



Note

By default, `armar` always creates a symbol table when you create a library of object files.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

To create a symbol table in a library that was created using the `-n` option, enter:

```
armar -s mylib.a
```

Related information

-n on page 847
--zs on page 856

5.1.24 --show_cmdline (armar)

Outputs the `armar` command line.

Syntax

```
--show_cmdline
```

Parameters

None.

Operation

Shows the command line after processing by the tool, and can be useful to check:

- The command line a build system is using.
- How the tool 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`).

Example

To show how `armar` processes the command-line options for the replacement of file obj1.o in `mylib.a`, enter:

```
> armar --show_cmdline -r mylib.a obj1.o
[armar --show_cmdline -r mylib.a obj1.o]
```

Related information

--via=filename (armar) on page 854

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.1.25 --sizes

Lists the `Code`, `RO Data`, `RW Data`, `ZI Data`, and `Debug` sizes of each member in the library.

Syntax

```
--sizes
```

Parameters

None.

Example

The following example shows the sizes of `app_1.o` and `app_2.o` in `mylib.a`:

```
> armar --sizes mylib.a

Code    RO Data    RW data    ZI Data    Debug   Object Name
 464          0          0          0     8612   app_1.o
3356          0          0      10244    11848   app_2.o
3820          0          0      10244    20460   TOTAL
```

Related information

--entries on page 843
--zt on page 857
--info=topic[,topic,...] (fromelf) on page 806

5.1.26 -t

Prints a table of contents for the library.

Syntax

```
-t
```

Parameters

None.

Operation

The files specified by `<file_list>` are included in the written list. If `<file_list>` is not specified, all files in the library are included in the order of the archive.

Examples

To display the table of contents of `mylib.a`, enter:

```
> armar -t mylib.a

app_1.o
app_2.o
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

To list the table of contents of a library in verbose mode, enter:

```
> armar -tv mylib.a

rw-rw-rw-     0/     0   7512 Jun 22 11:19 2009 app_1.o (offset    736)
rw-rw-rw-     0/     0   1452 May 19 16:25 2009 app_2.o (offset   8308)
```

Related information

-v (armar) on page 853
file_list on page 845

5.1.27 -T

Enables truncation of filenames when extracted files have library names that are longer than the file system can support.

Syntax

```
-T
```

Parameters

None.

Operation

By default, extracting a file with a name that is too long is an error. A diagnostic message is written and the file is not extracted.

Be aware that if multiple files in the library have the same truncated name, each subsequent file that is extracted overwrites the previously extracted file with that name. To prevent this, use the -c option.

Related information

-C (armar) on page 838
-x (armar) on page 856

5.1.28 -u (armar)

Updates older files in the specified archive.

Syntax

```
-u
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

When used with the `-r` option, files in the library are replaced only if the corresponding file has a modification time that is at least as new as the modification time of the file within library.

Related information

--new_files_only on page 847
-r on page 848

5.1.29 -v (armar)

Gives verbose output.

Syntax

```
-v
```

Parameters

None.

Operation

The output depends on what other options are used:

`-d`, `-r`, `-x`
: Write a detailed file-by-file description of the library creation, the constituent files, and maintenance activity.

`-p`
: Writes the name of the file to the standard output before writing the file itself to the `stdout`.

`-t`
: Includes a long listing of information about the files within the library.

`-x`
: Prints the filename preceding each extraction.

Related information

-d (armar) on page 840
-p on page 847
-r on page 848
-t on page 851
-x (armar) on page 856

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.1.30 --version_number (armar)

Displays the version of Arm® Compiler for Embedded tool that you are using.

Syntax

```
--version_number
```

Parameters

None.

Operation

The librarian displays the version number in the format `<Mmmuuxx>`, where:

- `<M>` is the major version number, `6`.
- `<mm>` is the minor version number.
- `<uu>` is the update number.
- `<xx>` is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions.

Related information

--help (armar) on page 845
--vsn (armar) on page 854

5.1.31 --via=filename (armar)

Reads an additional list of input filenames and tool options from `<filename>`.

Syntax

```
--via=<filename>
```

Parameters

`<filename>`

The name of a via file containing options to be included on the command line.

Operation

You can enter multiple `--via` options on the `armasm`, `armlink`, `fromelf`, and `armar` command lines. You can also include the `--via` options within a via file.

Related information

Via File Syntax on page 1082
Via file syntax rules on page 1083

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.1.32 --vsn (armar)

Displays the version information and the license details.

Syntax

```
--vsn
```

Parameters

None.

Operation

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm® Compiler for Embedded you are using. If you need to access the version in other tools or scripts, for example in build scripts, use the output from `--version_number`.

Table 5-1: Arm Compiler for Embedded Version Information

Field	Description
`Product: <Product Name>`	The product name, which depends on the currently activated license or suite that contains Arm Compiler for Embedded as a component, such as: • Arm *Software Success Kit* (SSK) with an SSK license entitlement. • Arm Development Studio. • Arm® Keil® *Microprocessor Development Kit* (MDK). For example, in Arm Development Studio, `--vsn` lists the product as: `Arm Development Studio <edition> Edition <version>`
`Component: <component>`	The version of Arm Compiler for Embedded. The version is made up of the major, minor, and update number: `Arm Compiler for Embedded <Major Version>.<Minor Version>.<Update Version>`
`Tool: <tool name> [<tool identifier>]`	The tool name and tool identifier. The tool identifier must not be used to identify versions of Arm Compiler for Embedded. The tool identifier is different for each operating system.

Example: armar --vsn output for Arm Development Studio

The `armar` output for Arm Compiler for Embedded 6.23 on Linux x86 with Arm Development Studio Gold Edition 2024.0:

```
Product: Arm Development Studio Gold Edition 2024.0
Component: Arm Compiler for Embedded 6.23
Tool: armar [<toolid>]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--help (armar) on page 845
--version_number (armar) on page 853
Arm Success Kits

5.1.33 -x (armar)

Extracts the files specified in `<file_list>` from the library to the current directory.

Syntax

```
-x
```

Parameters

None.

Operation

The contents of the library are not changed. If no file operands are given, all files in the library are extracted.

Be aware that if the name of a file in the library is longer than the file system can support, an error is displayed and the file is not extracted. To extract files with long filenames, use the `-T` option to truncate the names of files that are too long.

The files are extracted to the current location.

Example

To extract the files `file1.o` and `file2.o` from the `mylib.a` library in the directory `C:\temp` to `C:\temp\obj`, enter:

```
C: cd \temp\obj
armar -x ..\mylib.a file1.o,file2.o
```

Related information

-C (armar) on page 838
-T on page 852
file_list on page 845

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

5.1.34 --zs

Displays the symbol table for all files in the library.

Syntax

```
--zs
```

Parameters

None.

Example

To list the symbol table in `mylib.a`, enter:

```
> armar --zs mylib.a
__ARM_use_no_argv    from hello.o    at offset    412
main                 from hello.o    at offset    412
__ARM_use_no_argv    from test.o     at offset   7960
main                 from test.o     at offset   7960
__ARM_use_no_argv    from hello_ltcg.o at offset  11408
main                 from hello_ltcg.o at offset  11408
__ARM_use_no_argv    from h1.o       at offset  18532
main                 from h1.o       at offset  18532
__ARM_use_no_argv    from fncalls.o  at offset   2072
add                  from fncalls.o  at offset   2072
main                 from fncalls.o  at offset   2072
get_stacksize        from get_stacksize.o at offset   9672
altstack             from get_stacksize.o at offset   9672
__ARM_use_no_argv    from s.o        at offset  13068
main                 from s.o        at offset  13068
altstack             from s.o        at offset  13068
_Z1fv                from t.o        at offset  17064
_ZN1T1fEi            from t.o        at offset  17064
```

Related information

-n on page 847
-s (armar) on page 849

5.1.35 --zt

Lists both the member sizes and entry points for all files in the library.

Syntax

```
--zt
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

To list the member sizes and entry points for all files in `mylib.a`, enter:

```
> armar --zt mylib.a

      Code    RO Data    RW Data    ZI Data      Debug   Object Name
       838          0          0          0          0   hello.o
        16          0          0          0       2869   fncalls.o
       893          0          0          0          0   test.o
       962          0          0          0          0   get_stacksize.o
       838          0          0          0          0   hello_ltcg.o
         8          0          0         80          0   s.o
        56          0         50          0          0   strcopy.o
         4          0         44          0        168   emit-relocs-1a.o
        36          8          0          0         84   t.o
       838          0          0          0          0   h1.o
      4489          8         94         80       3121   TOTAL

ENTRY at offset 0 in section StrCopy of strcopy.o
ENTRY at offset 0 in section StrCopy of emit-relocs-1a.o
```

Related information

--entries on page 843
--sizes on page 850

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6. armasm Legacy Assembler Reference

A list of the command-line options for the `armasm` command, and other reference information that is relevant to `armasm`.


Note

The `armasm` legacy assembler is deprecated, and it has not been updated since Arm® Compiler 6.10. Also, `armasm` does not support:

- Armv8.4-A or later architectures.
- Certain backported options in Armv8.2-A and Armv8.3-A.
- Assembling `SVE` instructions.
- Armv8.1-M or later architectures, including MVE.
- All versions of the Armv8-R architecture.

As a reminder, `armasm` always reports the deprecation warning `A1950W`. To suppress this message, specify the `--diag_suppress=1950` option.

6.1 armasm Command-line Options

Describes the `armasm` command-line syntax and command-line options.


Note

The `armasm` legacy assembler is deprecated, and it has not been updated since Arm® Compiler 6.10. Also, `armasm` does not support:

- Armv8.4-A or later architectures.
- Certain backported options in Armv8.2-A and Armv8.3-A.
- Assembling `SVE` instructions.
- Armv8.1-M or later architectures, including MVE.
- All versions of the Armv8-R architecture.

As a reminder, `armasm` always reports the deprecation warning `A1950W`. To suppress this message, specify the `--diag_suppress=1950` option.

6.1.1 --16

Instructs `armasm` to interpret instructions as T32 instructions using the pre-UAL T32 syntax.

Syntax

```
--16
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Restrictions

Not supported for AArch64 state.

Operation

This option is equivalent to a `CODE16` directive at the head of the source file. Use the `--thumb` option to specify T32 instructions using the UAL syntax.

Related information

--thumb on page 894
CODE16 directive on page 981

6.1.2 --32

A synonym for the `--arm` command-line option.

Syntax

```
--32
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Related information

--arm on page 862

6.1.3 --apcs=qualifier...qualifier

Controls interworking and position independence when generating code.

Syntax

```
--apcs=<qualifier> ... <qualifier>
```

Parameters

`<qualifier>...<qualifier>`

A list of qualifiers. There must be:

- At least one qualifier present.
- No spaces or commas separating individual qualifiers in the list.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Each instance of `<qualifier>` must be one of:

none

Specifies that the input file does not use AAPCS. AAPCS registers are not set up. Other qualifiers are not permitted if you use `none`.

/interwork, /nointerwork

For Arm®v7-A, Armv7-R, Armv8-A, and Armv8-R, `/interwork` specifies that the code in the input file can interwork between A32 and T32 safely.

The default is `/interwork` for AArch32 targets that support both A32 and T32 instruction sets.

The default is `/nointerwork` for AArch32 targets that only support the T32 instruction set (M-profile targets).

When assembling for AArch64 state, interworking is not available.

/inter, /nointer

Are synonyms for `/interwork` and `/nointerwork`.

/ropi, /noropi

`/ropi` specifies that the code in the input file is *Read-Only Position Independent* (ROPI). The default is `/noropi`.

/pic, /nopic

Are synonyms for `/ropi` and `/noropi`.

/rwpi, /norwpi

`/rwpi` specifies that the code in the input file is *Read/Write Position Independent* (RWPI). The default is `/norwpi`.

/pid, /nopid

Are synonyms for `/rwpi` and `/norwpi`.

/fpic, /nofpic

`/fpic` specifies that the code in the input file is read-only independent and references to addresses are suitable for use in a Linux shared object. The default is `/nofpic`.

/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 you use `--apcs`. If floating-point support is not permitted (for example, because `--fpu=none` is specified, or because of other means), then `/hardfp` and `/softfp` are ignored. If floating-point support is permitted and the softfp calling convention is used (`--fpu=softvfp` or `--fpu=softvfp+fp-armv8`), then `/hardfp` gives an error.

`/softfp` is not supported for AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

This option specifies whether you are using the *Procedure Call Standard for the Arm Architecture* (AAPCS). It can also specify some attributes of code sections.

The AAPCS forms part of the *Base Standard Application Binary Interface for the Arm Architecture* (BSABI) specification. By writing code that adheres to the AAPCS, you can ensure that separately compiled and assembled modules can work together.


Note

AAPCS qualifiers do not affect the code produced by `armasm`. They are an assertion by the programmer that the code in the input file complies with a particular variant of AAPCS. They cause attributes to be set in the object file produced by `armasm`. The linker uses these attributes to check compatibility of files, and to select appropriate library variants.

Example

```
armasm --cpu=8-A.32 --apcs=/inter/hardfp inputfile.s
```

Related information

Procedure Call Standard for the Arm Architecture
Application Binary Interface (ABI)

6.1.4 --arm

Instructs `armasm` to interpret instructions as A32 instructions. It does not, however, guarantee A32-only code in the object file. This is the default. Using this option is equivalent to specifying the `ARM` or `CODE32` directive at the start of the source file.

Syntax

```
--arm
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Related information

--32 on page 860
--arm_only on page 862
ARM or CODE32 directive on page 978

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.1.5 --arm_only

Instructs `armasm` to only generate A32 code. This is similar to `--arm` but also has the property that `armasm` does not permit the generation of any T32 code.

Syntax

```
--arm_only
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Related information

--arm on page 862

6.1.6 --bi

A synonym for the `--bigend` command-line option.

Syntax

```
--bi
```

Parameters

None.

Related information

--bigend on page 863
--littleend on page 886

6.1.7 --bigend

Generates code suitable for an Arm® processor using big-endian memory access.

Default

The default is `--littleend`.

Syntax

```
--bigend
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Related information

--littleend on page 886
--bi on page 863

6.1.8 --brief_diagnostics, --no_brief_diagnostics

Enables and disables the output of brief diagnostic messages.

Syntax

```
--brief_diagnostics
--no_brief_diagnostics
```

Parameters

None.

Operation

This option instructs the assembler whether to use a shorter form of the diagnostic output. In this form, the original source line is not displayed and the error message text is not wrapped when it is too long to fit on a single line. The default is `--no_brief_diagnostics`.

Related information

--diag_error=tag[,tag,...] (armasm) on page 872
--diag_warning=tag[,tag,...] (armasm) on page 875

6.1.9 --checkreglist

Instructs the `armasm` to check `RLIST`, `LDM`, and `STM` register lists to ensure that all registers are provided in increasing register number order.

Syntax

```
--checkreglist
```

Parameters

None.

Operation

When this option is used, `armasm` gives a warning if the registers are not listed in order.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential



Note

In AArch32 state, this option is deprecated. Use `--diag_warning 1206` instead. In AArch64 state, this option is not supported.

Related information

--diag_warning=tag[,tag,...] (armasm) on page 875

6.1.10 --cpreproc

Instructs `armasm` to call `armclang` to preprocess the input file before assembling it.

Syntax

```
--cpreproc
```

Parameters

None.

Restrictions

You must use `--cpreproc_opts` with this option to correctly configure the `armclang` compiler for pre-processing.

`armasm` only passes the following command-line options to `armclang` by default:

- Basic pre-processor configuration options, such as `-E`.
- User specified include directories, `-I` directives.
- Anything specified in `--cpreproc_opts`.

Related information

--cpreproc_opts=option[,option,...] on page 865
Using the C preprocessor on page 943
-x (armclang) on page 229
Command-line options for preprocessing assembly source code

6.1.11 --cpreproc_opts=option[,option,...]

Enables `armasm` to pass options to `armclang` when using the C preprocessor.

Syntax

```
--cpreproc_opts=<option>[,<option>,...]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<option>[,<option>,...]

A comma-separated list of C preprocessing options.

At least one option must be specified.

Restrictions

As a minimum, you must specify the `armclang` options `--target` and either `-mcpu` or `-march` in `--cpreproc_opts`.

To assemble code containing C directives that require the C preprocessor, the input assembly source filename must have an upper-case extension `.s`.

You cannot pass the `armclang` option `-x assembler-with-cpp`, because it gets added to `armclang` after the source file name.



Note

Ensure that you specify compatible architectures in the `armclang` options `--target`, `-mcpu` or `-march`, and the `armasm` option `--cpu`.

Example

The options to the preprocessor in this example are `--cpreproc_opts=--target=arm-arm-none-eabi,-mcpu=cortex-a9,-D,DEF1,-D,DEF2`.

```
armasm --cpu=cortex-a9 --cpreproc --cpreproc_opts=--target=arm-arm-none-eabi,-
mcpu=cortex-a9,-D,DEF1,-D,DEF2 -I /path/to/includes1 -I /path/to/includes2 input.S
```

Related information

--cpreproc on page 865
Using the C preprocessor on page 943
-march on page 123
-mcpu on page 150
--target on page 221
-x (armclang) on page 229
Command-line options for preprocessing assembly source code
Mandatory armclang options

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.1.12 --cpu=list (armasm)

Lists the architecture and processor names that are supported by the `--cpu=name` option.

Syntax

```
--cpu=list
```

Parameters

`list`

List the supported architecture and processor names.

Related information

--cpu=name (armasm) on page 867

6.1.13 --cpu=name (armasm)

Enables code generation for the selected Arm® processor or architecture.

Default

There is no default option for `--cpu`.

Syntax

```
--cpu=<name>
```

Parameters

`<name>`

The name of a processor or architecture:

- Processor and architecture names are not case-sensitive.
- Wildcard characters are not accepted.

The following table shows the supported architectures. For a complete list of the supported architecture and processor names, specify the `--cpu=list` option.

Table 6-1: Supported Arm architectures

Architecture name	Description
`6-M`	Armv6 architecture microcontroller profile.
`6S-M`	Armv6 architecture microcontroller profile with OS extensions.
`7-A`	Armv7 architecture application profile.
`7-A.security`	Armv7-A architecture profile with the Security Extension and includes the `SMC` instruction (formerly `SMI`).
`7-R`	Armv7 architecture real-time profile.
`7-M`	Armv7 architecture microcontroller profile.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Architecture name	Description
7E-M	Armv7-M architecture profile with DSP extension.
8-A.32	Armv8-A architecture profile, AArch32 state.
8-A.32.crypto	Armv8-A architecture profile, AArch32 state with cryptographic instructions.
8-A.32.no_neon	Armv8-A architecture profile, AArch32 state without Advanced SIMD instructions.
8-A.64	Armv8-A architecture profile, AArch64 state.
8-A.64.crypto	Armv8-A architecture profile, AArch64 state with cryptographic instructions.
8.1-A.32	Armv8.1, for Armv8-A architecture profile, AArch32 state.
8.1-A.32.crypto	Armv8.1, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
8.1-A.64	Armv8.1, for Armv8-A architecture profile, AArch64 state.
8.1-A.64.crypto	Armv8.1, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
8.2-A.32	Armv8.2, for Armv8-A architecture profile, AArch32 state.
8.2-A.32.crypto	Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
8.2-A.32.crypto.dotprod	Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions and the VSDOT and VUDOT instructions.
8.2-A.32.dotprod	Armv8.2, for Armv8-A architecture profile, AArch32 state with the VSDOT and VUDOT instructions.
8.2-A.64	Armv8.2, for Armv8-A architecture profile, AArch64 state.
8.2-A.64.crypto	Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
8.2-A.64.crypto.dotprod	Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions and the SDOT and UDOT instructions.
8.2-A.64.dotprod	Armv8.2, for Armv8-A architecture profile, AArch64 state with the SDOT and UDOT instructions.
8.3-A.32	Armv8.3, for Armv8-A architecture profile, AArch32 state.
8.3-A.32.crypto	Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
8.3-A.32.crypto.dotprod	Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions and the VSDOT and VUDOT instructions.
8.3-A.32.dotprod	Armv8.3, for Armv8-A architecture profile, AArch32 state with the VSDOT and VUDOT instructions.
8.3-A.64	Armv8.3, for Armv8-A architecture profile, AArch64 state.
8.3-A.64.crypto	Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
8.3-A.64.crypto.dotprod	Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions and the SDOT and UDOT instructions.
8.3-A.64.dotprod	Armv8.3, for Armv8-A architecture profile, AArch64 state with the SDOT and UDOT instructions.
8-R	Armv8-R architecture profile, AArch32 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Architecture name	Description
`8-R.crypto`	Armv8-R architecture profile, AArch32 state with cryptographic instructions.
`8-R.no_neon`	Armv8-R architecture profile, AArch32 state without Advanced SIMD instructions.


Note

The full list of supported architectures and processors depends on your license.

Restrictions

`--cpu` has the following restrictions:

- You cannot specify both a processor and an architecture on the same command line.
- `armasm` does not support:
  - Armv8.4-A and later architectures.
  - Armv8-R AArch64 targets.
  - Certain backported options in Armv8.2-A and Armv8.3-A.
  - Assembling *M-profile Vector Extension* (MVE) or *Scalable Vector Extension* (SVE) instructions.

Operation

The following general points apply to processor and architecture options:

Processors

- Selecting the processor selects the appropriate architecture, *Floating-Point Unit* (FPU), and memory organization.
- If you specify a processor for the `--cpu` option, the generated code is optimized for that processor. Specifying a processor enables the assembler to use specific coprocessors or instruction scheduling for optimum performance.

Architectures

If you specify an architecture name for the `--cpu` option, the generated code can run on any processor supporting that architecture. For example, `--cpu=7-A` produces code that can be used by the Cortex®-A9 processor.

FPU

- Some specifications of `--cpu` imply an `--fpu` selection.


Note

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

- If no `--fpu` option is specified and the `--cpu` option does not imply an `--fpu` selection, then `--fpu=softvfp` is used.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A32/T32

- Specifying a processor or architecture that supports T32 instructions, such as `--cpu=cortex-a9`, does not make the assembler generate T32 code. It only enables features of the processor to be used, such as long multiply. Use the `--thumb` option to generate T32 code, unless the processor only supports T32 instructions.


Note

Specifying the target processor or architecture might make the generated object code incompatible with other Arm processors. For example, A32 code generated for architecture Armv8 might not run on a Cortex-A9 processor, if the generated object code includes instructions specific to Armv8. Therefore, you must choose the lowest common denominator processor suited to your purpose.

- If the architecture only supports T32, you do not have to specify `--thumb` on the command line. For example, if building for Cortex-M4 or Armv7-M with `--cpu=7-M`, you do not have to specify `--thumb` on the command line, because Armv7-M only supports T32. Similarly, Armv6-M and other T32-only architectures.

Example

```
armasm --cpu=Cortex-A17 inputfile.s
```

Related information

--apcs=qualifier...qualifier on page 860
--cpu=list (armasm) on page 866
--fpu=name (armasm) on page 882
--thumb on page 894
--unsafe on page 894
Arm Architecture Reference Manuals

6.1.14 --debug

Instructs the assembler to generate DWARF debug tables.

Default

The default is DWARF 3.

Syntax

```
--debug
```

Parameters

None.

Operation

`--debug` is a synonym for `-g`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

Local symbols are not preserved with `--debug`. You must specify `--keep` if you want to preserve the local symbols to aid debugging.

Related information

--dwarf2 on page 876
--dwarf3 on page 877
--keep (armasm) on page 884
-g (armasm) on page 882

6.1.15 --depend=dependfile

Writes makefile dependency lines to a file.

Syntax

```
--depend=<dependfile>
```

Parameters

<dependfile>

The file to contain the makefile dependency lines.

Operation

Source file dependency lists are suitable for use with make utilities.

Related information

--md on page 888
--depend_format=string on page 871

6.1.16 --depend_format=string

Specifies the format of output dependency files, for compatibility with some UNIX make programs.

Syntax

```
--depend_format=<string>
```

Parameters

<string>

One of:

unix

Generates dependency file entries using UNIX-style path separators.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

unix_escaped
The same as `unix`, but escapes spaces with `\`.

unix_quoted
The same as `unix`, but surrounds path names with double quotes.

Related information
--depend=dependfile on page 871

6.1.17 --diag_error=tag[,tag,...] (armasm)

Sets diagnostic messages that have a specific tag to Error severity.

Syntax
```
--diag_error=<tag>[,<tag>,...]
```

Parameters
<tag>
One of:

- 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.

Operation
Diagnostic messages output by the assembler can be identified by a tag in the form of `{\<prefix>\}<number>`, where the `<prefix>` is A.

You can specify more than one tag with this option by separating each tag using a comma. You can specify the optional assembler prefix `A` before the tag number. If any prefix other than A is included, the message number is ignored.

The following table shows the meaning of the term severity used in the option descriptions:

Table 6-2: Severity of diagnostic messages

Severity	Description
Error	Errors indicate violations in the syntactic or semantic rules of assembly language. Assembly continues, but object code is not generated.
Warning	Warnings indicate unusual conditions in your code that might indicate a problem. Assembly continues, and object code is generated unless any problems with an Error severity are detected.
Remark	Remarks indicate common, but not recommended, use of assembly language. These diagnostics are not issued by default. Assembly continues, and object code is generated unless any problems with an Error severity are detected.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--brief_diagnostics, --no_brief_diagnostics on page 864
--diag_remark=tag[,tag,...] (armasm) on page 873
--diag_suppress=tag[,tag,...] (armasm) on page 874
--diag_warning=tag[,tag,...] (armasm) on page 875

6.1.18 --diag_remark=tag[,tag,...] (armasm)

Sets diagnostic messages that have a specific tag to Remark severity.

Syntax

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

Parameters

<tag>

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.

Operation

Diagnostic messages output by the assembler can be identified by a tag in the form of {\<prefix>\}<number>, where the <prefix> is A.

You can specify more than one tag with this option by separating each tag using a comma. You can specify the optional assembler prefix A before the tag number. If any prefix other than A is included, the message number is ignored.

Related information

--brief_diagnostics, --no_brief_diagnostics on page 864
--diag_error=tag[,tag,...] (armasm) on page 872
--diag_suppress=tag[,tag,...] (armasm) on page 874
--diag_warning=tag[,tag,...] (armasm) on page 875

6.1.19 --diag_style=arm|ide|gnu (armasm)

Specifies the display style for diagnostic messages.

Default

The default is --diag_style=arm.

Syntax

```
--diag_style=<string>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<string>

One of:

arm

Display messages using the legacy 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`.

Operation

`--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 information

--brief_diagnostics, --no_brief_diagnostics on page 864

6.1.20 --diag_suppress=tag[,tag,...] (armasm)

Suppresses diagnostic messages that have a specific tag.



Reducing the severity of diagnostic messages might prevent the tool from reporting important faults. Arm recommends that you do not reduce the severity of diagnostics unless you understand the impact on your software.

Syntax

```
--diag_suppress=<tag>[,<tag>,...]
```

Parameters

<tag>

One of:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- 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.

Diagnostic messages output by `armasm` can be identified by a tag in the form of `{\<prefix>\}<number>`, where the `<prefix>` is A.

You can specify more than one tag with this option by separating each tag using a comma.

Example

For example, to suppress the warning messages that have numbers `1293` and `187`, use the following command:

```
armasm --cpu=8-A.64 --diag_suppress=1293,187
```

You can specify the optional assembler prefix `A` before the tag number. For example:

```
armasm --cpu=8-A.64 --diag_suppress=A1293,A187
```

If any prefix other than A is included, the message number is ignored. Diagnostic message tags can be cut and pasted directly into a command line.

Related information

--brief_diagnostics, --no_brief_diagnostics on page 864
--diag_error=tag[,tag,...] (armasm) on page 872
--diag_remark=tag[,tag,...] (armasm) on page 873
--diag_warning=tag[,tag,...] (armasm) on page 875

6.1.21 --diag_warning=tag[,tag,...] (armasm)

Sets diagnostic messages that have a specific tag to Warning severity.

Syntax

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

Parameters

`<tag>`

One of the following:

- 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.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Diagnostic messages output by the assembler can be identified by a tag in the form of *prefix*<number>, where the <prefix> is A.

You can specify more than one tag with this option by separating each tag using a comma.

You can specify the optional assembler prefix A before the tag number. If any prefix other than A is included, the message number is ignored.

Related information

--brief_diagnostics, --no_brief_diagnostics on page 864
--diag_error=tag[,tag,...] (armasm) on page 872
--diag_remark=tag[,tag,...] (armasm) on page 873
--diag_suppress=tag[,tag,...] (armasm) on page 874

6.1.22 --dllexport_all

Controls symbol visibility when building DLLs.

Syntax

```
--dllexport_all
```

Parameters

None.

Operation

This option gives all exported global symbols STV_PROTECTED visibility in ELF rather than STV_HIDDEN, unless overridden by source directives.

Related information

EXPORT or GLOBAL directive on page 993

6.1.23 --dwarf2

Uses DWARF 2 debug table format.

Syntax

```
--dwarf2
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operation

This option can be used with `--debug`, to instruct `armasm` to generate DWARF 2 debug tables.

Related information

--debug on page 870
--dwarf3 on page 877

6.1.24 --dwarf3

Uses DWARF 3 debug table format.

Syntax

```
--dwarf3
```

Parameters

None.

Operation

This option can be used with `--debug`, to instruct the assembler to generate DWARF 3 debug tables. This option is the default if `--debug` is specified.

Related information

--debug on page 870
--dwarf2 on page 876

6.1.25 --errors=errorfile

Redirects the output of diagnostic messages from stderr to the specified errors file.

Syntax

```
-\\-errors=<errorfile>
```

Parameters

`<errorfile>`

The file to contain the diagnostic messages.

6.1.26 --exceptions, --no_exceptions (armasm)

Enables or disables exception handling.

Default

The default is `--no_exceptions`, which causes no tables to be generated.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
--exceptions
--no_exceptions
```

Parameters

None.

Operation

These options instruct `armasm` to switch on or off exception table generation for all functions defined by `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Related information

--exceptions_unwind, --no_exceptions_unwind on page 878
FRAME UNWIND ON directive on page 1004
FRAME UNWIND OFF directive on page 1004
FUNCTION or PROC directive on page 1005
ENDFUNC or ENDP directive on page 991

6.1.27 --exceptions_unwind, --no_exceptions_unwind

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

Default

The default is `--exceptions_unwind`.

Syntax

```
--exceptions_unwind
--no_exceptions_unwind
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Operation

For finer control, use the `FRAME UNWIND ON` and `FRAME UNWIND OFF` directives.

Related information

--exceptions, --no_exceptions (armasm) on page 877
FRAME UNWIND ON directive on page 1004
FRAME UNWIND OFF directive on page 1004

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

FUNCTION or PROC directive on page 1005
ENDFUNC or ENDP directive on page 991

6.1.28 --execstack, --no_execstack

Generates a `.note.GNU-stack` section marking the stack as either executable or non-executable.

Syntax

```
--execstack
--no_execstack
```

Parameters

None.

Operation

You can also use the `AREA` directive to generate either an executable or non-executable `.note.GNU-stack` section. The following code generates an executable `.note.GNU-stack` section. Omitting the `CODE` attribute generates a non-executable `.note.GNU-stack` section.

```
    AREA    |.note.GNU-stack|,ALIGN=0,READONLY,NOALLOC,CODE
```

In the absence of `--execstack` and `--no_execstack`, the `.note.GNU-stack` section is not generated unless it is specified by the `AREA` directive.

If both the command-line option and source directive are used and are different, then the stack is marked as executable.

Table 6-3: Specifying a command-line option and an AREA directive for GNU-stack sections

	`--execstack` command-line option	`--no_execstack` command-line option
execstack `AREA` directive	execstack	execstack
no_execstack `AREA` directive	execstack	no_execstack

Related information

AREA directive on page 974

6.1.29 --execute_only

Adds the `EXECONLY AREA` attribute to all code sections.

Syntax

```
--execute_only
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Restrictions

This option is only supported for:

- Processors that support the Arm®v8-M mainline or Armv8-M Baseline architecture.
- Processors that support the Armv7-M architecture, such as Cortex®-M3, Cortex-M4, and Cortex-M7.
- Processors that support the Armv6-M architecture. However, execute-only is not supported on Armv6-M for any form of position independent code.

Operation

The `EXECONLY AREA` attribute causes the linker to treat the section as *eXecute-Only* (XO).

It is your responsibility to ensure that the code in the section is safe to run in XO memory. For example:

- The code must not contain literal pools.
- The code must not attempt to load data from the same, or another, XO section.

Related information

AREA directive on page 974

6.1.30 --fpmode=model

Specifies floating-point standard conformance and sets library attributes and floating-point optimizations.

Syntax

```
--fpmode=<model>
```

Parameters

`<model>`

One of:

`none`

Source code is not permitted to use any floating-point type or floating-point instruction. This option overrides any explicit `--fpu=<name>` option.

`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.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

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

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

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

Finite values are as predicted by the IEEE standard. It is not guaranteed that NaNs and infinities are produced in all circumstances defined by the IEEE model, or that when they are produced, they have the same sign. Also, it is not guaranteed that the sign of zero is that predicted by the IEEE model.

fast
Some value altering optimizations, where accuracy is sacrificed to fast execution. This is not IEEE compatible, and is not standard C.


Note

This option does not cause any changes to the code that you write.

Example

```
armasm --cpu=8-A.32 --fpmode ieee_full inputfile.s
```

Related information

--fpu=name (armasm) on page 882
IEEE Standards Association

6.1.31 --fpu=list (armasm)

Lists the FPU architecture names that are supported by the `--fpu=name` option.

Syntax

```
--fpu=list
```

Parameters

list
List the supported FPU architecture names.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

```
armasm --fpu=list
```

Related information

--fpmode=model on page 880
--fpu=name (armasm) on page 882

6.1.32 --fpu=name (armasm)

Specifies the target FPU architecture.

Syntax

```
--fpu=<name>
```

Parameters

<name>

The name of the target FPU architecture. Specify `--fpu=list` to list the supported FPU architecture names that you can use with `--fpu=<name>`.

The default floating-point architecture depends on the target architecture.

Restrictions

Software floating-point linkage is not available for AArch64 state.

Operation

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. Floating-point instructions also produce either errors or warnings if assembled for the wrong target FPU.

`armasm` sets a build attribute corresponding to name in the object file. The linker determines compatibility between object files, and selection of libraries, accordingly.

Related information

--fpmode=model on page 880

6.1.33 -g (armasm)

Enables the generation of debug tables.

Syntax

```
-g
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

Operation

This option is a synonym for `--debug`.

Related information

--debug on page 870

6.1.34 --help (armasm)

Displays a summary of the main command-line options.

Default

This is the default if you specify the tool command without any options or source files.

Syntax

```
--help
```

Parameters

None.

Related information

--version_number (armasm) on page 896
--vsn (armasm) on page 897

6.1.35 -idir[,dir, ...]

Adds directories to the source file include path.

Syntax

```
-i<dir>[,<dir>, ...]
```

Parameters

`<dir>`

The directory to add to the source file include path.

Operation

Any directories added using this option have to be fully qualified.

Related information

GET or INCLUDE directive on page 1007

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.1.36 --keep (armasm)

Instructs the assembler to keep named local labels in the symbol table of the object file, for use by the debugger.

Syntax

```
--keep
```

Parameters

None.

Related information

KEEP directive on page 1013

6.1.37 --length=n

Sets the listing page length.

Default

The default is 66 lines.

Syntax

```
--length=<n>
```

Parameters

<n>

The listing page length.

Length zero means an unpaged listing.

Related information

--list=file on page 885

6.1.38 --li

A synonym for the `--littleend` command-line option.

Syntax

```
--li
```

Parameters

None.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--littleend on page 886
--bigend on page 863

6.1.39 --library_type=lib (armasm)

Enables the selected library to be used at link time.

Default

The default is `standardlib`.

Syntax

```
--library_type=<lib>
```

Parameters

`<lib>`

One of:

`standardlib`

Specifies that the full Arm® runtime libraries are selected at link time.

`microlib`

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



- This option can be used with the compiler, assembler, or linker when use of the libraries require more specialized optimizations.
- This option can be overridden at link time by providing it to the linker.
- microlib is not supported for AArch64 state.

Related information

Building an application with microlib

6.1.40 --list=file

Instructs the assembler to output a detailed listing of the assembly language produced by the assembler to a file.

Syntax

```
--list=<file>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<file>

The name of the file to contain the assembly.

Operation

If - is given as <file>, the listing is sent to stdout.

Use the following command-line options to control the behavior of --list:

- --no_terse.
- --width.
- --length.
- --xref.

Related information

--no_terse on page 890
--width=n on page 898
--length=n on page 884
--xref on page 898
OPT directive on page 1021

6.1.41 --list=

Instructs the assembler to send the detailed assembly language listing to inputfile.lst.

Syntax

```
--list=
```

Parameters

None.

Operation

You can use --list without the equals sign and filename to send the output to inputfile.lst. However, this syntax is deprecated and the assembler issues a warning. This syntax is to be removed in a later release. Use --list= instead.

Related information

--list=file on page 885

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.1.42 --littleend

Generates code suitable for an Arm® processor using little-endian memory access.

Default

The default is `--littleend`.

Syntax

```
--littleend
```

Parameters

None.

Related information

--bigend on page 863
--li on page 884

6.1.43 -m (armasm)

Instructs the assembler to write source file dependency lists to `stdout`.

Syntax

```
-m
```

Parameters

None.

Related information

--md on page 888

6.1.44 --maxcache=n

Sets the maximum source cache size in bytes.

Default

The default is 8MB.

Syntax

```
--maxcache=<n>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<n>

The maximum source cache size in bytes.

Operation

`armasm` gives a warning if the size is less than 8MB.

6.1.45 --md

Creates makefile dependency lists.

Syntax

```
--md
```

Parameters

None.

Operation

This option instructs the assembler to write source file dependency lists to `inputfile.d`.

Related information

-m (armasm) on page 887

6.1.46 --no_code_gen

Instructs the assembler to exit after pass 1, generating no object file. This option is useful if you only want to check the syntax of the source code or directives.

Syntax

```
--no_code_gen
```

Parameters

None.

6.1.47 --no_esc

Instructs the assembler to ignore C-style escaped special characters, such as `\n` and `\t`.

Syntax

```
--no_esc
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

None.

6.1.48 --no_hide_all

Gives all exported and imported global symbols `STV_DEFAULT` visibility in ELF rather than `STV_HIDDEN`, unless overridden using source directives.

Syntax

```
--no_hide_all
```

Parameters

None.

Operation

You can use the following directives to specify an attribute that overrides the implicit symbol visibility:

- `EXPORT`.
- `EXTERN`.
- `GLOBAL`.
- `IMPORT`.

Related information

EXPORT or GLOBAL directive on page 993
IMPORT and EXTERN directives on page 1010

6.1.49 --no_regs

Instructs `armasm` not to predefine register names.

Syntax

```
--no_regs
```

Parameters

None.

Operation

This option is deprecated. In AArch32 state, use `--regnames=none` instead.

Related information

--regnames on page 892

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.1.50 --no_terse

Instructs the assembler to show in the list file the lines of assembly code that it has skipped because of conditional assembly.

Syntax

```
--no_terse
```

Parameters

None.

Operation

If you do not specify this option, the assembler does not output the skipped assembly code to the list file.

This option turns off the terse flag. By default the terse flag is on.

Related information

--list=file on page 885

6.1.51 --no_warn

Turns off warning messages.

Syntax

```
--no_warn
```

Parameters

None.

Related information

--diag_warning=tag[,tag,...] (armasm) on page 875

6.1.52 -o filename (armasm)

Specifies the name of the output file.

Default

The default output filename is `<inputfilename>.o`.

Syntax

```
-o <filename>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<filename>
The output filename.

Operation

If this option is not used, the assembler creates an object filename in the form `<inputfilename>.o`. This option is case-sensitive.

6.1.53 --pd

A synonym for the `--predefine` command-line option.

Syntax

```
--pd
```

Parameters

None.

Related information

--predefine "directive" on page 891

6.1.54 --predefine "directive"

Instructs `armasm` to pre-execute one of the `SETA`, `SETL`, or `SETS` directives.

Syntax

```
--predefine "<directive>"
```

Parameters

"<directive>"
The directive to pre-execute.

Operation

You must enclose `<directive>` in quotes, for example:

```
armasm --cpu=8-A.64 --predefine "VariableName SETA 20" inputfile.s
```

`armasm` also executes a corresponding `GBLL`, `GBLS`, or `GBLA` directive to define the variable before setting its value.

The variable name is case-sensitive. The variables defined using the command line are global to `armasm` source files specified on the command line.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Considerations when using `--predefine`

Be aware of the following:

- The command-line interface of your system might require you to enter special character combinations, such as `\"`, to include strings in `<directive>`. Alternatively, you can use `--via <file>` to include a `--predefine` argument. The command-line interface does not alter arguments from `--via` files.
- `--predefine` is not equivalent to the compiler option `-D<name>`. `--predefine` defines a global variable whereas `-D<name>` defines a macro that the C preprocessor expands.

  Although you can use predefined global variables in combination with assembly control directives, for example `IF` and `ELSE` to control conditional assembly, they are not intended to provide the same functionality as the C preprocessor in `armasm`. If you require this functionality, we recommend that you use the compiler to pre-process your assembly code.

Related information

--pd on page 891
GBLA, GBLL, and GBLS directives on page 1006
IF, ELSE, ENDIF, and ELIF directives on page 1008
SETA, SETL, and SETS directives on page 1028

6.1.55 --regnames

Controls the predefinition of register names.

Syntax

```
--regnames=<option>
```

Parameters

`<option>`

One of the following:

`none`

Instructs `armasm` not to predefine register names.

`callstd`

Defines additional register names based on the AAPCS variant that you are using, as specified by the `--apcs` option.

`all`

Defines all AAPCS registers regardless of the value of `--apcs`.

Restrictions

Not supported for AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

--no_regs on page 889
--apcs=qualifier...qualifier on page 860

6.1.56 --report-if-not-wysiwyg

Instructs `armasm` to report when it outputs an encoding that was not directly requested in the source code.

Syntax

```
--report-if-not-wysiwyg
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Operation

`armasm` reports the output of an encoding that was not directly requested in the source code when it:

- Uses a pseudo-instruction that is not available in other assemblers, for example `MOV32`.
- Outputs an encoding that does not directly match the instruction mnemonic, for example if the assembler outputs the `MVN` encoding when assembling the `MOV` instruction.
- Inserts additional instructions where necessary for instruction syntax semantics, for example `armasm` can insert a missing `IT` instruction before a conditional T32 instruction.

6.1.57 --show_cmdline (armasm)

Outputs the `armasm` command line.

Syntax

```
--show_cmdline
```

Parameters

None.

Operation

Shows the command line after processing by the tool, and can be useful to check:

- The command line a build system is using.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- How the tool 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 information

--via=filename (armasm) on page 896

6.1.58 --thumb

Instructs `armasm` to interpret instructions as T32 instructions, using UAL syntax. This is equivalent to a `THUMB` directive at the start of the source file.

Syntax

```
--thumb
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Related information

--arm on page 862
THUMB directive on page 1030

6.1.59 --unaligned_access, --no_unaligned_access (armasm)

Enables or disables unaligned accesses to data on Arm-based processors.

Syntax

```
--unaligned_access
--no_unaligned_access
```

Parameters

None.

Operation

These options instruct the assembler to set an attribute in the object file to enable or disable the use of unaligned accesses.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.1.60 --unsafe

Enables instructions for other architectures to be assembled without error.

Syntax

```
--unsafe
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Operation

This option downgrades error messages to corresponding warning messages. It also suppresses warnings about operator precedence.

Related information

--diag_error=tag[,tag,...] (armasm) on page 872
--diag_warning=tag[,tag,...] (armasm) on page 875
Binary operators on page 961

6.1.61 --untyped_local_labels

Causes `armasm` not to set the T32 bit for the address of a numeric local label referenced in an `LDR` pseudo-instruction.

Syntax

```
--untyped_local_labels
```

Parameters

None.

Restrictions

Not supported for AArch64 state.

Operation

When this option is not used, if you reference a numeric local label in an `LDR` pseudo-instruction, and the label is in T32 code, then `armasm` sets the T32 bit (bit 0) of the address. You can then use the address as the target for a `BX` or `BLX` instruction.

If you require the actual address of the numeric local label, without the T32 bit set, then use this option.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

When using this option, if you use the address in a branch (register) instruction, `armasm` treats it as an A32 code address, causing the branch to arrive in A32 state, meaning it would interpret this code as A32 instructions.

Example

```
    THUMB
    ...
1
    ...
    LDR r0,=%B1 ; r0 contains the address of numeric local label "1",
                ; T32 bit is not set if --untyped_local_labels was used
    ...
```

Related information

Numeric local labels on page 953

6.1.62 --version_number (armasm)

Displays the version of armasm that you are using.

Syntax

```
--version_number
```

Parameters

None.

Operation

The assembler displays the version number in the format `<M><mm><uu><xx>`, where:

- `<M>` is the major version number, `6`.
- `<mm>` is the minor version number.
- `<uu>` is the update number.
- `<xx>` is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions.

6.1.63 --via=filename (armasm)

Reads an additional list of input filenames and tool options from `<filename>`.

Syntax

```
--via=<filename>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<filename>

The name of a via file containing options to be included on the command line.

Operation

You can enter multiple `--via` options on the `armasm`, `armlink`, `fromelf`, and `armar` command lines. You can also include the `--via` options within a via file.

Related information

Via File Syntax on page 1082
Via file syntax rules on page 1083

6.1.64 --vsn (armasm)

Displays the version information and the license details.

Syntax

```
--vsn
```

Parameters

None.

Operation

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm® Compiler for Embedded tool you are using. If you need to access the version in other tools or scripts, for example in build scripts, use the output from `--version_number`.

Table 6-4: Arm Compiler for Embedded Version Information

Field	Description
`Product: <Product Name>`	The product name, which depends on the currently activated license or suite that contains Arm Compiler for Embedded as a component, such as: • Arm *Software Success Kit* (SSK) with an SSK license entitlement. • Arm Development Studio. • Arm® Keil® *Microprocessor Development Kit* (MDK). For example, in Arm Development Studio, `--vsn` lists the product as: `Arm Development Studio <edition> Edition <version>`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Field	Description
`Component: <component>`	The version of Arm Compiler for Embedded. The version is made up of the major, minor, and update number: `Arm Compiler for Embedded <Major Version>.<Minor Version>.<Update Version>`
`Tool: <tool name> [<tool identifier>]`	The tool name and tool identifier. The tool identifier must not be used to identify versions of Arm Compiler for Embedded. The tool identifier is different for each operating system.

Example: armasm --vsn output for Arm Development Studio

The `armasm` output for Arm Compiler for Embedded 6.23 on Linux x86 with Arm Development Studio Gold Edition 2024.0:

```
Warning: A1950W: The legacy armasm assembler is deprecated. Consider using the
 armclang integrated assembler instead.
Product: Arm Development Studio Gold Edition 2024.0
Component: Arm Compiler for Embedded 6.23
Tool: armasm [<toolid>]
```

6.1.65 --width=n

Sets the listing page width.

Default

The default is 79 characters.

Syntax

```
--width=<n>
```

Parameters

<n>

The listing page width.

Related information

--list=file on page 885

6.1.66 --xref

Instructs the assembler to list cross-referencing information on symbols, including where they were defined and where they were used, both inside and outside macros.

Default

The default is off.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Syntax

```
--xref
```

Parameters

None.

Related information

--list=file on page 885

6.2 Structure of armasm Assembly Language Modules

Describes the structure of `armasm` assembly language source files.

6.2.1 Syntax of source lines in armasm syntax assembly language

The `armasm` assembler parses and assembles armasm syntax assembly language to produce object code.

Syntax

Each line of `armasm` syntax assembly language source code has this general form:

```
{<symbol>} {<instruction>|<directive>|<pseudo-instruction>} ;{<comment>}
```

All three sections of the source line are optional.

`<symbol>` is usually a label. In instructions and pseudo-instructions it is always a label. In some directives it is a symbol for a variable or a constant. The description of the directive makes this clear in each case.

`<symbol>` must begin in the first column. It cannot contain any white space character such as a space or a tab unless it is enclosed by bars (|).

Labels are symbolic representations of addresses. You can use labels to mark specific addresses that you want to refer to from other parts of the code. Numeric local labels are a subclass of labels that begin with a number in the range 0-99. Unlike other labels, a numeric local label can be defined many times. This makes them useful when generating labels with a macro.

Directives provide important information to the assembler that either affects the assembly process or affects the final output image.

Instructions and pseudo-instructions make up the code a processor uses to perform tasks.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Note

Instructions, pseudo-instructions, and directives must be preceded by white space, such as a space or a tab, irrespective of whether there is a preceding label or not. Some directives do not allow the use of a label.

A comment is the final part of a source line. The first semicolon on a line marks the beginning of a comment except where the semicolon appears inside a string literal. The end of the line is the end of the comment. A comment alone is a valid line. The assembler ignores all comments. You can use blank lines to make your code more readable.

Note

Comments inside source files and header files that are provided by Arm® might not be accurate and must not be treated as documentation about the product.

Considerations when writing armasm syntax language source code

You must write instruction mnemonics, pseudo-instructions, directives, and symbolic register names (except `a1-a4` and `v1-v8` in A32 or T32 instructions) in either all uppercase or all lowercase. You must not use mixed case. Labels and comments can be in uppercase, lowercase, or mixed case.

```
        AREA     A32ex, CODE, READONLY

                                ; Name this block of code A32ex

        ENTRY                   ; Mark first instruction to execute
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0x123456      ; AArch32 semihosting (formerly SWI)
        END                     ; Mark end of file
```

To make source files easier to read, you can split a long line of source into several lines by placing a backslash character (\) at the end of the line. The backslash must not be followed by any other characters, including spaces and tabs. The assembler treats the backslash followed by end-of-line sequence as white space. You can also use blank lines to make your code more readable.

Note

Do not use the backslash followed by end-of-line sequence within quoted strings.

The limit on the length of lines, including any extensions using backslashes, is 4095 characters.

Related information

Literals on page 901

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Labels on page 951
Numeric local labels on page 953
String literals on page 955
Symbol naming rules on page 947
Syntax of numeric literals on page 957

6.2.2 Literals

armasm syntax language source code can contain numeric, string, Boolean, and single character literals.

Literals can be expressed as:

- Decimal numbers, for example 123.
- Hexadecimal numbers, for example 0x7B.
- Numbers in any base from 2 to 9, for example 5_204 is a number in base 5.
- Floating point numbers, for example 123.4.
- Boolean values {TRUE} or {FALSE}.
- Single character values enclosed by single quotes, for example 'w'.
- Strings enclosed in double quotes, for example "This is a string".


Note

In most cases, a string containing a single character is accepted as a single character value. For example ADD r0,r1,#"a" is accepted, but ADD r0,r1,#"ab" is faulted.

You can also use variables and names to represent literals.

Related information

Syntax of source lines in armasm syntax assembly language on page 899

6.2.3 ELF sections and the AREA directive

Object files produced by the armasm assembler are divided into sections. In armasm syntax assembly source code, you use the AREA directive to mark the start of a section.

ELF sections are independent, named, indivisible sequences of code or data. A single code section is the minimum required to produce an application.

The output of an assembly or compilation can include:

- One or more code sections. These are usually read-only sections.
- One or more data sections. These are usually read-write sections. They might be zero-initialized (ZI).

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The linker places each section in a program image according to section placement rules. Sections that are adjacent in source files are not necessarily adjacent in the application image

Use the `AREA` directive to name the section and set its attributes. The attributes are placed after the name, separated by commas.

You can choose any name for your sections. However, names starting with any non-alphabetic character must be enclosed in bars, or an `AREA name missing` error is generated. For example, `|1_DataArea|`.

The following example defines a single read-only section called `A32ex` that contains code:

```
    AREA  A32ex, CODE, READONLY ; Name this block of code A32ex
```

Related information

An example armasm syntax assembly language module on page 902
AREA directive on page 974
Scatter-loading Features on page 643

6.2.4 An example armasm syntax assembly language module

An armasm syntax assembly language module has several constituent parts.

These are:

- ELF sections (defined by the `AREA` directive).
- Application entry (defined by the `ENTRY` directive).
- Application execution.
- Application termination.
- Program end (defined by the `END` directive).

Constituents of an A32 assembly language module

The following example defines a single section called `A32ex` that contains code and is marked as being `READONLY`. This example uses the A32 instruction set.

```
        AREA     A32ex, CODE, READONLY

                                ; Name this block of code A32ex
        ENTRY                   ; Mark first instruction to execute
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0x123456      ; AArch32 semihosting (formerly SWI)
        END                     ; Mark end of file
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Constituents of an A64 assembly language module

The following example defines a single section called `A64ex` that contains code and is marked as being `READONLY`. This example uses the A64 instruction set.

```
        AREA     A64ex, CODE, READONLY

                                ; Name this block of code A64ex
        ENTRY                   ; Mark first instruction to execute
start
        MOV      w0, #10        ; Set up parameters
        MOV      w1, #3
        ADD      w0, w0, w1     ; w0 = w0 + w1
stop
        MOV      x1, #0x26
        MOVK     x1, #2, LSL #16
        STR      x1, [sp,#0]    ; ADP_Stopped_ApplicationExit
        MOV      x0, #0
        STR      x0, [sp,#8]    ; Exit status code
        MOV      x1, sp         ; x1 contains the address of parameter block
        MOV      w0, #0x18      ; angel_SWIreason_ReportException
        HLT      0xf000         ; AArch64 semihosting
        END                     ; Mark end of file
```

Constituents of a T32 assembly language module

The following example defines a single section called `T32ex` that contains code and is marked as being `READONLY`. This example uses the T32 instruction set.

```
        AREA     T32ex, CODE, READONLY

                                ; Name this block of code T32ex
        ENTRY                   ; Mark first instruction to execute
        THUMB
start
        MOV      r0, #10        ; Set up parameters
        MOV      r1, #3
        ADD      r0, r0, r1     ; r0 = r0 + r1
stop
        MOV      r0, #0x18      ; angel_SWIreason_ReportException
        LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
        SVC      #0xab          ; AArch32 semihosting (formerly SWI)
        ALIGN    4              ; Aligned on 4-byte boundary
        END                     ; Mark end of file
```

Application entry

The `ENTRY` directive declares an entry point to the program. It marks the first instruction to be executed. In applications using the C library, an entry point is also contained within the C library initialization code. Initialization code and exception handlers also contain entry points.

Application execution in A32 or T32 code

The application code begins executing at the label `start`, where it loads the decimal values 10 and 3 into registers `R0` and `R1`. These registers are added together and the result placed in `R0`.

Application execution in A64 code

The application code begins executing at the label `start`, where it loads the decimal values 10 and 3 into registers `W0` and `W1`. These registers are added together and the result placed in `W0`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Application termination

After executing the main code, the application terminates by returning control to the debugger.

A32 and T32 code

You do this in A32 and T32 code using the semihosting SVC instruction:

- In A32 code, the semihosting SVC instruction is `0x123456` by default.
- In T32 code, use the semihosting SVC instruction is `0xAB` by default.

A32 and T32 code uses the following parameters:

- R0 equal to `angel_SWIreason_ReportException` (`0x18`).
- R1 equal to `ADP_Stopped_ApplicationExit` (`0x20026`).

A64 code

In A64 code, use `HLT` instruction `0xF000` to invoke the semihosting interface.

A64 code uses the following parameters:

- W0 equal to `angel_SWIreason_ReportException` (`0x18`).
- X1 is the address of a block of two parameters. The first is the exception type, `ADP_Stopped_ApplicationExit` (`0x20026`) and the second is the exit status code.

Program end

The `END` directive instructs the assembler to stop processing this source file. Every assembly language source module must finish with an `END` directive on a line by itself. Any lines following the `END` directive are ignored by the assembler.

Related information

END directive on page 990
ENTRY directive on page 991
ELF sections and the AREA directive on page 901
Semihosting for AArch32 and AArch64

6.3 Writing A32/T32 Instructions in armasm Syntax Assembly Language

`armasm` supports A32 and T32 instructions and the use of macros in the armasm syntax assembly language.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.3.1 About the Unified Assembler Language

Unified Assembler Language (UAL) is a common syntax for A32 and T32 instructions. It supersedes earlier versions of both the A32 and T32 assembler languages.

Code that is written using UAL can be assembled for A32 or T32 for any Arm® processor. `armasm` faults the use of unavailable instructions.

`armasm` can assemble code that is written in pre-UAL and UAL syntax.

By default, `armasm` expects source code to be written in UAL. `armasm` accepts UAL syntax if any of the directives `CODE32`, `ARM`, or `THUMB` is used or if you assemble with any of the `--32`, `--arm`, or `--thumb` command-line options. `armasm` also accepts source code that is written in pre-UAL A32 assembly language when you assemble with the `CODE32` or `ARM` directive.

`armasm` accepts source code that is written in pre-UAL T32 assembly language when you assemble using the `--16` command-line option, or the `CODE16` directive in the source code.

Note

The pre-UAL T32 assembly language does not support 32-bit T32 instructions.

Related information

--16 on page 859
ARM or CODE32 directive on page 978
CODE16 directive on page 981
THUMB directive on page 1030
--32 on page 860
--arm on page 862
--thumb on page 894

6.3.2 Syntax differences between UAL and A64 assembly language

Unified Assembler Language (UAL) is the assembler syntax that is used by the A32 and T32 instruction sets. A64 assembly language is the assembler syntax that is used by the A64 instruction set.

Arm®v7 supports UAL for A32 and T32 instruction sets. Armv8 supports UAL for A32 and T32 instruction sets, and A64 assembly language for the A64 instruction set.

The general statement format and operand order of A64 assembly language is the same as UAL, but there are some differences between them. The following table describes the main differences:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 6-5: Syntax differences between UAL and A64 assembly language

UAL	A64
You make an instruction conditional by appending a condition code suffix directly to the mnemonic, with no delimiter. For example: `BEQ label`	For conditionally executed instructions, you separate the condition code suffix from the mnemonic using a `.` delimiter.For example: `B.EQ label`
Apart from the `IT` instruction, there are no unconditionally executed integer instructions that use a condition code as an operand.	A64 provides several unconditionally executed instructions that use a condition code as an operand. For these instructions, you specify the condition code to test for in the final operand position. For example: `CSEL w1,w2,w3,EQ`
The `.W` and `.N` instruction width specifiers control whether the assembler generates a 32-bit or 16-bit encoding for a T32 instruction.	A64 is a fixed width 32-bit instruction set so does not support `.W` and `.N` qualifiers.
The core register names are R0-R15.	Qualify register names to indicate the operand data size, either 32-bit (W0-W31) or 64-bit (X0-X31).
You can refer to registers R13, R14, and R15 as synonyms for SP, LR, and PC respectively.	In AArch64, there is no register that is named W31 or X31. Instead, you can refer to register 31 as SP, WZR, or XZR, depending on the context. You cannot refer to PC either by name or number. LR is an alias for register 30.
A32 has no equivalent of the extend operators.	You can specify an extend operator in several instructions to control how a portion of the second source register value is sign or zero extended. For example, in the following instruction, `UXTB` is the extend type (zero extend,byte) and `#2` is an optional left shift amount: `ADD X1, X2, W3, UXTB #2`

6.3.3 Register usage in subroutine calls

You use branch instructions to call and return from subroutines. The Procedure Call Standard for the Arm® Architecture defines how to use registers in subroutine calls.

A subroutine is a block of code that performs a task based on some arguments and optionally returns a result. By convention, you use registers R0 to R3 to pass arguments to subroutines, and R0 to pass a result back to the callers. A subroutine that requires more than four inputs uses the stack for the additional inputs.

To call subroutines, use a branch and link instruction. The syntax is:

```
BL  destination
```

where `<destination>` is usually the label on the first instruction of the subroutine.

`<destination>` can also be a PC-relative expression.

The `BL` instruction:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Places the return address in the link register.
- Sets the PC to the address of the subroutine.

After the subroutine code has executed you can use a `BX LR` instruction to return.


Note

Calls between separately assembled or compiled modules must comply with the restrictions and conventions defined by the *Procedure Call Standard for the Arm Architecture*.

Example

The following example shows a subroutine, `doadd`, that adds the values of two arguments and returns a result in `R0`:

```
        AREA    subrout, CODE, READONLY     ; Name this block of code

        ENTRY                       ; Mark first instruction to execute
start   MOV     r0, #10             ; Set up parameters
        MOV     r1, #3
        BL      doadd               ; Call subroutine
stop    MOV     r0, #0x18           ; angel_SWIreason_ReportException
        LDR     r1, =0x20026        ; ADP_Stopped_ApplicationExit
        SVC     #0x123456           ; AArch32 semihosting (formerly SWI)
doadd   ADD     r0, r0, r1          ; Subroutine code
        BX      lr                  ; Return from subroutine
        END                         ; Mark end of file
```

Related information

Stack operations for nested subroutines on page 921
Procedure Call Standard for the Arm Architecture
Procedure Call Standard for the Arm 64-bit Architecture (AArch64)

6.3.4 Load immediate values

To represent some immediate values, you might have to use a sequence of instructions rather than a single instruction.

A32 and T32 instructions can only be 32 bits wide. You can use a `MOV` or `MVN` instruction to load a register with an immediate value from a range that depends on the instruction set. Certain 32-bit values cannot be represented as an immediate operand to a single 32-bit instruction, although you can load these values from memory in a single instruction.

You can load any 32-bit immediate value into a register with two instructions, a `MOV` followed by a `MOVT`. Or, you can use a pseudo-instruction, `MOV32`, to construct the instruction sequence for you.

You can also use the `LDR` pseudo-instruction to load immediate values into a register.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

You can include many commonly-used immediate values directly as operands within data processing instructions, without a separate load operation. The range of immediate values that you can include as operands in 16-bit T32 instructions is much smaller.

Related information

Load immediate values using MOV and MVN on page 908
Load immediate values using MOV32 on page 910
Load immediate values using LDR Rd, =const on page 911
LDR pseudo-instruction on page 1038

6.3.5 Load immediate values using MOV and MVN

The `MOV` and `MVN` instructions can write a range of immediate values to a register.

In A32:

- `MOV` can load any 8-bit immediate value, giving a range of {`0x0`,...,`0xFF`} ({0,...,255}).

  It can also rotate these values by any even number.

  These values are also available as immediate operands in many data processing operations, without being loaded in a separate instruction.
- `MVN` can load the bitwise complements of these values. The numerical values are `-(<n>+1)`, where `<n>` is the value available in `MOV`.
- `MOV` can load any 16-bit number, giving a range of {`0x0`,...,`0xFFFF`} ({0,...,65535}).

The following table shows the range of 8-bit values that can be loaded in a single A32 `MOV` or `MVN` instruction (for data processing operations). The value to load must be a multiple of the second value shown in the Decimal column.

Table 6-6: A32 state immediate values (8-bit)

Binary	Decimal	Hexadecimal	MVN value [a]	Notes
`000000000000000000000000abcdefgh`	{0,1,...,255}	{0,...,FF}	{-1,...,-256}	-
`00000000000000000000abcdefgh00`	{0,4,...,1020}	{0,...,3FC}	{-1,...,-1021}	-
`0000000000000000000abcdefgh0000`	{0,16,...,4080}	{0,...,FF0}	{-1,...,-4081}	-
`00000000000000000abcdefgh000000`	{0,64,...,16320}	{0,...,3FC0}	{-1,...,-16321}	-
...	...	...	...	-
`abcdefgh000000000000000000000000`	{0,1,...,255} x 2^{24}	{0-FF000000}	{1,...,256} x -2^{24} - 1	-
`cdefgh000000000000000000000000ab`	(bit pattern)	-	(bit pattern)	See [b] in Note
`efgh000000000000000000000000abcd`	(bit pattern)	-	(bit pattern)	See [b] in Note
`gh000000000000000000000000abcdef`	(bit pattern)	-	(bit pattern)	See [b] in Note

The following table shows the range of 16-bit values that can be loaded in a single `MOV` A32 instruction:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 6-7: A32 state immediate values in MOV instructions

Binary	Decimal	Hexadecimal	MVN value	Notes
0000000000000000abcdefghijklmnop	{0,1,...,65535}	{0,...,FFFF}	-	See [c] in Note


Note

These notes give extra information on both tables.

[a]

The MVN values are only available directly as operands in MVN instructions.

[b]

These values are available in A32 only. All the other values in this table are also available in 32-bit T32 instructions.

[c]

These values are not available directly as operands in other instructions.

In T32:

- The 32-bit MOV instruction can load:
  - Any 8-bit immediate value, giving a range of 0x0,...,0xFF (0,...,255).
  - Any 8-bit immediate value, shifted left by any number.
  - Any 8-bit pattern duplicated in all four bytes of a register.
  - Any 8-bit pattern duplicated in bytes 0 and 2, with bytes 1 and 3 set to 0.
  - Any 8-bit pattern duplicated in bytes 1 and 3, with bytes 0 and 2 set to 0.

  These values are also available as immediate operands in many data processing operations, without being loaded in a separate instruction.
- The 32-bit MVN instruction can load the bitwise complements of these values. The numerical values are –(<n>+1), where <n> is the value available in MOV.
- The 32-bit MOV instruction can load any 16-bit number, giving a range of 0x0,...,0xFFFF (0-65535). These values are not available as immediate operands in data processing operations.

In architectures with T32, the 16-bit T32 MOV instruction can load any immediate value in the range 0-255.

The following table shows the range of values that can be loaded in a single 32-bit T32 MOV or MVN instruction (for data processing operations). The value to load must be a multiple of the second value shown in the Decimal column.

Table 6-8: 32-bit T32 immediate values

Binary	Decimal	Hexadecimal	MVN value [a]	Notes
000000000000000000000000abcdefgh	{0,1,...,255}	{0x0,...,0xFF}	{-1,...,-256}	-
000000000000000000000000abcdefgh0	{0,2,...,510}	{0x0,...,0x1FE}	{-1,...,-511}	-
000000000000000000000000abcdefgh00	{0,4,...,1020}	{0x0,...,0x3FC}	{-1,...,-1021}	-
...	...	...	...	-

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Binary	Decimal	Hexadecimal	MVN value [a]	Notes
0abcdefgh00000000000000000000000	{0,1,...,255} x 2^{23}	{0x0, ...,0x7F800000}	{1,...,256} x -2^{23} - 1	-
abcdefgh00000000000000000000000	{0,1,...,255} x 2^{24}	{0x0, ...,0xFF000000}	{1,...,256} x -2^{24} - 1	-
abcdefghabcdefghabcdefghabcdefgh	(bit pattern)	0xXYXYXYXY	0xXYXYXYXY	-
00000000abcdefgh00000000abcdefgh	(bit pattern)	0x00XY00XY	0xFFXYFFXY	-
abcdefgh00000000abcdefgh00000000	(bit pattern)	0xXY00XY00	0xXYFFXYFF	-
00000000000000000000abcdefghijkl	{0,1,...,4095}	{0x0,...,0xFFF}	-	See [b] in Note

The following table shows the range of 16-bit values that can be loaded by the MOV 32-bit T32 instruction:

Table 6-9: 32-bit T32 immediate values in MOV instructions

Binary	Decimal	Hexadecimal	MVN value	Notes
0000000000000000abcdefghijklmnop	{0,1,...,65535}	{0x0,...,0xFFFF}	-	See [c] in Note

Note

These notes give extra information on the tables.

[a]

The MVN values are only available directly as operands in MVN instructions.

[b]

These values are available directly as operands in ADD, SUB, and MOV instructions, but not in MVN or any other data processing instructions.

[c]

These values are only available in MOV instructions.

In both A32 and T32, you do not have to decide whether to use MOV or MVN. The assembler uses whichever is appropriate. This is useful if the value is an assembly-time variable.

If you write an instruction with an immediate value that is not available, the assembler reports the error: `Immediate n out of range for this operation.`

Related information

Load immediate values on page 907

6.3.6 Load immediate values using MOV32

To load any 32-bit immediate value, a pair of MOV and MOVT instructions is equivalent to a MOV32 pseudo-instruction.

Both A32 and T32 instruction sets include:

- A MOV instruction that can load any value in the range 0x00000000 to 0x0000FFFF into a register.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- A MOVT instruction that can load any value in the range 0x0000 to 0xFFFF into the most significant half of a register, without altering the contents of the least significant half.

You can use these two instructions to construct any 32-bit immediate value in a register. Alternatively, you can use the MOV32 pseudo-instruction. The assembler generates the MOV, MOVT instruction pair for you.

You can also use the MOV32 instruction to load addresses into registers by using a label or any PC-relative expression in place of an immediate value. The assembler puts a relocation directive into the object file for the linker to resolve the address at link-time.

Related information

Register-relative and PC-relative expressions on page 950
MOV32 pseudo-instruction on page 1040

6.3.7 Load immediate values using LDR Rd, =const

The LDR Rd,=const pseudo-instruction generates the most efficient single instruction to load any 32-bit number.

You can use this pseudo-instruction to generate constants that are out of range of the MOV and MVN instructions.

The LDR pseudo-instruction generates the most efficient single instruction for the specified immediate value:

- If the immediate value can be constructed with a single MOV or MVN instruction, the assembler generates the appropriate instruction.
- If the immediate value cannot be constructed with a single MOV or MVN instruction, the assembler:
  - Places the value in a literal pool (a portion of memory embedded in the code to hold constant values).
  - Generates an LDR instruction with a PC-relative address that reads the constant from the literal pool.

  For example:

```
    LDR      rn, [pc, #offset to literal pool]

                        ; load register n with one word
                        ; from the address [pc + offset]
```

  You must ensure that there is a literal pool within range of the LDR instruction generated by the assembler.

Related information

Literal pools on page 912
LDR pseudo-instruction on page 1038

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.3.8 Literal pools

The assembler uses literal pools to store some constant data in code sections. You can use the LTORG directive to ensure a literal pool is within range.

The assembler places a literal pool at the end of each section. The end of a section is defined either by the END directive at the end of the assembly or by the AREA directive at the start of the following section. The END directive at the end of an included file does not signal the end of a section.

In large sections the default literal pool can be out of range of one or more LDR instructions. The offset from the PC to the constant must be:

- Less than 4KB in A32 or T32 code when the 32-bit LDR instruction is available, but can be in either direction.
- Forward and less than 1KB when only the 16-bit T32 LDR instruction is available.

When an `LDR Rd,=const` pseudo-instruction requires the immediate value to be placed in a literal pool, the assembler:

- Checks if the value is available and addressable in any previous literal pools. If so, it addresses the existing constant.
- Attempts to place the value in the next literal pool if it is not already available.

If the next literal pool is out of range, the assembler generates an error message. In this case you must use the LTORG directive to place an additional literal pool in the code. Place the LTORG directive after the failed LDR pseudo-instruction, and within the valid range for an LDR instruction.

You must place literal pools where the processor does not attempt to execute them as instructions. Place them after unconditional branch instructions, or after the return instruction at the end of a subroutine.

Example of placing literal pools

The following example shows the placement of literal pools. The instructions listed as comments are the A32 instructions generated by the armasm legacy assembler.

```
        AREA     Loadcon, CODE, READONLY

        ENTRY                       ; Mark first instruction to execute
start
        BL       func1              ; Branch to first subroutine
        BL       func2              ; Branch to second subroutine
stop
        MOV      r0, #0x18          ; angel_SWIreason_ReportException
        LDR      r1, =0x20026       ; ADP_Stopped_ApplicationExit
        SVC      #0x123456          ; AArch32 semihosting (formerly SWI)
func1
        LDR      r0, =42            ; => MOV R0, #42
        LDR      r1, =0x55555555    ; => LDR R1, [PC, #offset to
                                    ; Literal Pool 1]
        LDR      r2, =0xFFFFFFFF    ; => MVN R2, #0
        BX       lr
        LTORG                       ; Literal Pool 1 contains
                                    ; literal 0x55555555
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
func2
        LDR      r3, =0x55555555   ; => LDR R3, [PC, #offset to
                                   ; Literal Pool 1]
        ; LDR r4, =0x66666666      ; If this is uncommented it
                                   ; fails, because Literal Pool 2
                                   ; is out of reach
        BX       lr
LargeTable
        SPACE    4200              ; Starting at the current location,
                                   ; clears a 4200 byte area of memory
                                   ; to zero
        END                        ; Literal Pool 2 is inserted here,
                                   ; but is out of range of the LDR
                                   ; pseudo-instruction that needs it
```

Related information

LTORG directive on page 1015
Load immediate values using LDR Rd, =const on page 911

6.3.9 Load addresses into registers

It is often necessary to load an address into a register. There are several ways to do this.

For example, you might have to load the address of a variable, a string literal, or the start location of a jump table.

Addresses are normally expressed as offsets from a label, or from the current PC or other register.

You can load an address into a register either:

- Using the instruction `ADR`.
- Using the pseudo-instruction `ADRL`.
- Using the pseudo-instruction `MOV32`.
- From a literal pool using the pseudo-instruction `LDR Rd,=Label`.

Related information

Load addresses to a register using ADR on page 913
Load addresses to a register using ADRL on page 915
Load immediate values using MOV32 on page 910
Load addresses to a register using LDR Rd, =label on page 916

6.3.10 Load addresses to a register using ADR

The `ADR` instruction loads an address within a certain range, without performing a data load.

`ADR` accepts a PC-relative expression, that is, a label with an optional offset where the address of the label is relative to the PC.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential



Note

The label used with ADR must be within the same code section. The assembler faults references to labels that are out of range in the same section.

The available range of addresses for the ADR instruction depends on the instruction set and encoding:

A32

Any value that can be produced by rotating an 8-bit value right by any even number of bits within a 32-bit word. The range is relative to the PC.

32-bit T32 encoding

±4095 bytes to a byte, halfword, or word-aligned address.

16-bit T32 encoding

0 to 1020 bytes. <label> must be word-aligned. You can use the ALIGN directive to ensure this.

Example of a jump table implementation with ADR

This example shows A32 code that implements a jump table. Here, the ADR instruction loads the address of the jump table.

```
        AREA    Jump, CODE, READONLY ; Name this block of code


ARM                          ; Following code is A32 code
num     EQU     2                    ; Number of entries in jump table
        ENTRY                        ; Mark first instruction to execute
start                                ; First instruction to call
        MOV     r0, #0               ; Set up the three arguments
        MOV     r1, #3
        MOV     r2, #2
        BL      arithfunc            ; Call the function
stop
        MOV     r0, #0x18            ; angel_SWIreason_ReportException
        LDR     r1, =0x20026         ; ADP_Stopped_ApplicationExit

SVC    #0x123456             ; AArch32 semihosting (formerly SWI)
arithfunc                            ; Label the function
        CMP     r0, #num             ; Treat function code as unsigned
                                     ; integer
        BXHS    lr                   ; If code is >= num then return
        ADR     r3, JumpTable        ; Load address of jump table
        LDR     pc, [r3,r0,LSL#2]    ; Jump to the appropriate routine
JumpTable
        DCD     DoAdd
        DCD     DoSub
DoAdd
        ADD     r0, r1, r2           ; Operation 0
        BX      lr                   ; Return
DoSub
        SUB     r0, r1, r2           ; Operation 1
        BX      lr                   ; Return
        END                          ; Mark the end of this file
```

In this example, the function arithfunc takes three arguments and returns a result in R0. The first argument determines the operation to be carried out on the second and third arguments:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

argument1=0

Result = argument2 + argument3.

argument1=1

Result = argument2 - argument3.

The jump table is implemented with the following instructions and assembler directives:

EQU

Is an assembler directive. You use it to give a value to a symbol. In this example, it assigns the value 2 to `<num>`. When `<num>` is used elsewhere in the code, the value 2 is substituted. Using `EQU` in this way is similar to using `#define` to define a constant in C.

DCD

Declares one or more words of store. In this example, each `DCD` stores the address of a routine that handles a particular clause of the jump table.

LDR

The `LDR PC,[R3,R0,LSL#2]` instruction loads the address of the required clause of the jump table into the PC. It:

- Multiplies the clause number in `R0` by 4 to give a word offset.
- Adds the result to the address of the jump table.
- Loads the contents of the combined address into the PC.

Related information

Load addresses to a register using LDR Rd, =label on page 916
Load addresses to a register using ADRL on page 915

6.3.11 Load addresses to a register using ADRL

The `ADRL` pseudo-instruction loads an address within a certain range, without performing a data load. The range is wider than that of the `ADR` instruction.

`ADRL` accepts a PC-relative expression, that is, a label with an optional offset where the address of the label is relative to the current PC.



Note

The label used with `ADRL` must be within the same code section. The assembler faults references to labels that are out of range in the same section.

The assembler converts an `ADRL r<n>,<label>` pseudo-instruction by generating:

- Two data processing instructions that load the address, if it is in range.
- An error message if the address cannot be constructed in two instructions.

The available range depends on the instruction set and encoding.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A32

Any value that can be generated by two `ADD` or two `SUB` instructions. That is, any value that can be produced by the addition of two values, each of which is 8 bits rotated right by any even number of bits within a 32-bit word. The range is relative to the PC.

32-bit T32 encoding

±1MB to a byte, halfword, or word-aligned address.

16-bit T32 encoding

`ADRL` is not available.

Related information

Load addresses to a register using ADR on page 913
Load addresses to a register using LDR Rd, =label on page 916

6.3.12 Load addresses to a register using LDR Rd, =label

The `LDR Rd,=<label>` pseudo-instruction places an address in a literal pool and then loads the address into a register.

`LDR Rd,=<label>` can load any 32-bit numeric value into a register. It also accepts PC-relative expressions such as labels, and labels with offsets.

The assembler converts an `LDR Rd,=<label>` pseudo-instruction by:

- Placing the address of `<label>` in a literal pool (a portion of memory embedded in the code to hold constant values).
- Generating a PC-relative `LDR` instruction that reads the address from the literal pool, for example:

```
LDR rn [pc, #offset_to_literal_pool]
                  ; load register n with one word
                  ; from the address [pc + offset]
```

You must ensure that the literal pool is within range of the `LDR` pseudo-instruction that needs to access it.

Example of loading using LDR Rd, =label

The following example shows a section with two literal pools. The final `LDR` pseudo-instruction needs to access the second literal pool, but it is out of range. Uncommenting this line causes the assembler to generate an error.

The instructions listed in the comments are the A32 instructions generated by the assembler.

```
        AREA   LDRlabel, CODE, READONLY
        ENTRY                    ; Mark first instruction to execute
start
        BL     func1             ; Branch to first subroutine
        BL     func2             ; Branch to second subroutine
stop
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
        MOV     r0, #0x18           ; angel_SWIreason_ReportException
        LDR     r1, =0x20026        ; ADP_Stopped_ApplicationExit
        SVC     #0x123456           ; AArch32 semihosting (formerly SWI)
func1
        LDR     r0, =start          ; => LDR r0,[PC, #offset into Literal Pool 1]
        LDR     r1, =Darea + 12     ; => LDR r1,[PC, #offset into Literal Pool 1]
        LDR     r2, =Darea + 6000   ; => LDR r2,[PC, #offset into Literal Pool 1]
        BX      lr                  ; Return
        LTORG                       ; Literal Pool 1
func2
        LDR     r3, =Darea + 6000   ; => LDR r3,[PC, #offset into Literal Pool 1]
                                    ; (sharing with previous literal)
        ; LDR   r4, =Darea + 6004  ; If uncommented, produces an error because
                                    ; Literal Pool 2 is out of range.
        BX      lr                  ; Return
Darea   SPACE   8000                ; Starting at the current location, clears
                                    ; a 8000 byte area of memory to zero.
        END                         ; Literal Pool 2 is automatically inserted
                                    ; after the END directive.
                                    ; It is out of range of all the LDR
                                    ; pseudo-instructions in this example.
```

Example of string copy

The following example shows an A32 code routine that overwrites one string with another. It uses the LDR pseudo-instruction to load the addresses of the two strings from a data section. The following are particularly significant:

DCB

The DCB directive defines one or more bytes of store. In addition to integer values, DCB accepts quoted strings. Each character of the string is placed in a consecutive byte.

LDR, STR

The LDR and STR instructions use post-indexed addressing to update their address registers. For example, the instruction:

```
   LDRB    r2,[r1],#1
```

loads R2 with the contents of the address pointed to by R1 and then increments R1 by 1.

The example also shows how, unlike the ADR and ADRL pseudo-instructions, you can use the LDR pseudo-instruction with labels that are outside the current section. The assembler places a relocation directive in the object code when the source file is assembled. The relocation directive instructs the linker to resolve the address at link time. The address remains valid wherever the linker places the section containing the LDR and the literal pool.

```
        AREA    StrCopy, CODE, READONLY
        ENTRY                       ; Mark first instruction to execute
start
        LDR     r1, =srcstr         ; Pointer to first string
        LDR     r0, =dststr         ; Pointer to second string
        BL      strcopy             ; Call subroutine to do copy
stop
        MOV     r0, #0x18           ; angel_SWIreason_ReportException
        LDR     r1, =0x20026        ; ADP_Stopped_ApplicationExit
        SVC     #0x123456           ; AArch32 semihosting (formerly SWI)
strcopy
        LDRB    r2, [r1],#1         ; Load byte and update address
        STRB    r2, [r0],#1         ; Store byte and update address
        CMP     r2, #0              ; Check for zero terminator
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
        BNE     strcopy             ; Keep going if not
        MOV     pc,lr               ; Return
        AREA    Strings, DATA, READWRITE
srcstr  DCB     "First string - source",0
dststr  DCB     "Second string - destination",0
        END
```

Related information

Load addresses to a register using ADRL on page 915
Load immediate values using LDR Rd, =const on page 911
LDR pseudo-instruction on page 1038
DCB directive on page 984

6.3.13 Other ways to load and store registers

You can load and store registers using LDR, STR and MOV (register) instructions.

You can load any 32-bit value from memory into a register with an LDR data load instruction. To store registers into memory you can use the STR data store instruction.

You can use the MOV instruction to move any 32-bit data from one register to another.

Related information

Load and store multiple register instructions on page 918
Load and store multiple register instructions in A32 and T32 on page 919

6.3.14 Load and store multiple register instructions

The A32 and T32 instruction sets include instructions that load and store multiple registers. These instructions can provide a more efficient way of transferring the contents of several registers to and from memory than using single register loads and stores.

Multiple register transfer instructions are most often used for block copy and for stack operations at subroutine entry and exit. The advantages of using a multiple register transfer instruction instead of a series of single data transfer instructions include:

- Smaller code size.
- A single instruction fetch overhead, rather than many instruction fetches.
- On uncached Arm® processors, the first word of data transferred by a load or store multiple is always a nonsequential memory cycle, but all subsequent words transferred can be sequential memory cycles. Sequential memory cycles are faster in most systems.



Note

The lowest numbered register is transferred to or from the lowest memory address accessed, and the highest numbered register to or from the highest address accessed. The order of the registers in the register list in the instructions makes no

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

difference. You can use the `--diag_warning 1206` assembler command-line option to check that registers in register lists are specified in increasing order.

Related information

Load and store multiple register instructions in A32 and T32 on page 919
Stack implementation using LDM and STM on page 920
Stack operations for nested subroutines on page 921
Block copy with LDM and STM on page 922

6.3.15 Load and store multiple register instructions in A32 and T32

Instructions are available in both the A32 and T32 instruction sets to load and store multiple registers.

They are:

LDM
: Load Multiple registers.

STM
: Store Multiple registers.

PUSH
: Store multiple registers onto the stack and update the stack pointer.

POP
: Load multiple registers off the stack, and update the stack pointer.

In `LDM` and `STM` instructions:

- The list of registers loaded or stored can include:
  - In A32 instructions, any or all of R0-R12, SP, LR, and PC.
  - In 32-bit T32 instructions, any or all of R0-R12, and optionally LR or PC (`LDM` only) with some restrictions.
  - In 16-bit T32 instructions, any or all of R0-R7.
- The address must be word-aligned. It can be:
  - Incremented after each transfer.
  - Incremented before each transfer (A32 instructions only).
  - Decremented after each transfer (A32 instructions only).
  - Decremented before each transfer (not in 16-bit encoded T32 instructions).
- The base register can be either:
  - Updated to point to the next block of data in memory.
  - Left as it was before the instruction.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

When the base register is updated to point to the next block in memory, this is called writeback, that is, the adjusted address is written back to the base register.

In PUSH and POP instructions:

- The stack pointer (SP) is the base register, and is always updated.
- The address is incremented after each transfer in POP instructions, and decremented before each transfer in PUSH instructions.
- The list of registers loaded or stored can include:
  - In A32 instructions, any or all of R0-R12, SP, LR, and PC.
  - In 32-bit T32 instructions, any or all of R0-R12, and optionally LR or PC (POP only) with some restrictions.
  - In 16-bit T32 instructions, any or all of R0-R7, and optionally LR (PUSH only) or PC (POP only).


Note

Use of SP in the list of registers in these A32 instructions is deprecated.

A32 STM and PUSH instructions that use PC in the list of registers, and A32 LDM and POP instructions that use both PC and LR in the list of registers are deprecated.

Related information

Load and store multiple register instructions on page 918

6.3.16 Stack implementation using LDM and STM

You can use the LDM and STM instructions to implement pop and push operations respectively. You use a suffix to indicate the stack type.

The load and store multiple instructions can update the base register. For stack operations, the base register is usually the stack pointer, SP. This means that you can use these instructions to implement push and pop operations for any number of registers in a single instruction.

The load and store multiple instructions can be used with several types of stack:

Descending or ascending

The stack grows downwards, starting with a high address and progressing to a lower one (a descending stack), or upwards, starting from a low address and progressing to a higher address (an ascending stack).

Full or empty

The stack pointer can either point to the last item in the stack (a full stack), or the next free space on the stack (an empty stack).

To make it easier for the programmer, stack-oriented suffixes can be used instead of the increment or decrement, and before or after suffixes. The following table shows the stack-oriented suffixes and their equivalent addressing mode suffixes for load and store instructions:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 6-10: Stack-oriented suffixes and equivalent addressing mode suffixes

Stack-oriented suffix	For store or push instructions	For load or pop instructions
FD (Full Descending stack)	DB (Decrement Before)	IA (Increment After)
FA (Full Ascending stack)	IB (Increment Before)	DA (Decrement After)
ED (Empty Descending stack)	DA (Decrement After)	IB (Increment Before)
EA (Empty Ascending stack)	IA (Increment After)	DB (Decrement Before)

The following table shows the load and store multiple instructions with the stack-oriented suffixes for the various stack types:

Table 6-11: Suffixes for load and store multiple instructions

Stack type	Store	Load
Full descending	`STMFD` (`STMDB`, Decrement Before)	`LDMFD` (`LDM`, increment after)
Full ascending	`STMFA` (`STMIB`, Increment Before)	`LDMFA` (`LDMDA`, Decrement After)
Empty descending	`STMED` (`STMDA`, Decrement After)	`LDMED` (`LDMIB`, Increment Before)
Empty ascending	`STMEA` (`STM`, increment after)	`LDMEA` (`LDMDB`, Decrement Before)

For example:

```
STMFD    sp!, <r0-r5>  ; Push onto a Full Descending Stack
LDMFD    sp!, {r0-r5}  ; Pop from a Full Descending Stack
```


Note

The *Procedure Call Standard for the Arm Architecture* (AAPCS), and `armclang` always use a full descending stack.

The `PUSH` and `POP` instructions assume a full descending stack. They are the preferred synonyms for `STMDB` and `LDM` with writeback.

Related information

Load and store multiple register instructions on page 918
Procedure Call Standard for the Arm Architecture

6.3.17 Stack operations for nested subroutines

Stack operations can be very useful at subroutine entry and exit to avoid losing register contents if other subroutines are called.

At the start of a subroutine, any working registers required can be stored on the stack, and at exit they can be popped off again.

In addition, if the link register is pushed onto the stack at entry, additional subroutine calls can be made safely without causing the return address to be lost. If you do this, you can also return from a

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

subroutine by popping the PC off the stack at exit, instead of popping the LR and then moving that value into the PC. For example:

```
subroutine  PUSH    {r5-r7,lr} ; Push work registers and lr
            ; code
            BL      somewhere_else
            ; code
            POP     {r5-r7,pc} ; Pop work registers and pc
```

Related information

Register usage in subroutine calls on page 906
Load and store multiple register instructions on page 918
Procedure Call Standard for the Arm Architecture
Procedure Call Standard for the Arm 64-bit Architecture (AArch64)

6.3.18 Block copy with LDM and STM

You can sometimes make code more efficient by using `LDM` and `STM` instead of `LDR` and `STR` instructions.

Example of block copy without LDM and STM

The following example is an A32 code routine that copies a set of words from a source location to a destination a single word at a time:

```
        AREA  Word, CODE, READONLY  ; name the block of code

num     EQU   20                    ; set number of words to be copied
        ENTRY                       ; mark the first instruction called
start
        LDR   r0, =src              ; r0 = pointer to source block
        LDR   r1, =dst              ; r1 = pointer to destination block
        MOV   r2, #num              ; r2 = number of words to copy
wordcopy
        LDR   r3, [r0], #4          ; load a word from the source and
        STR   r3, [r1], #4          ; store it to the destination
        SUBS  r2, r2, #1            ; decrement the counter
        BNE   wordcopy              ; ... copy more
stop
        MOV   r0, #0x18             ; angel_SWIreason_ReportException
        LDR   r1, =0x20026          ; ADP_Stopped_ApplicationExit

SVC   #0x123456             ; AArch32 semihosting (formerly SWI)

        AREA  BlockData, DATA, READWRITE
src     DCD   1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst     DCD   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        END
```

You can make this module more efficient by using `LDM` and `STM` for as much of the copying as possible. Eight is a sensible number of words to transfer at a time, given the number of available

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

registers. You can find the number of eight-word multiples in the block to be copied (if R2 = number of words to be copied) using:

```
    MOVS    r3, r2, LSR #3    ; number of eight word multiples
```

You can use this value to control the number of iterations through a loop that copies eight words per iteration. When there are fewer than eight words left, you can find the number of words left (assuming that R2 has not been corrupted) using:

```
    ANDS    r2, r2, #7
```

Example of block copy using LDM and STM

The following example lists the block copy module rewritten to use LDM and STM for copying:

```
      AREA    Block, CODE, READONLY ; name this block of code

num   EQU     20                    ; set number of words to be copied
      ENTRY                         ; mark the first instruction called
start
      LDR     r0, =src              ; r0 = pointer to source block
      LDR     r1, =dst              ; r1 = pointer to destination block
      MOV     r2, #num              ; r2 = number of words to copy
      MOV     sp, #0x400            ; Set up stack pointer (sp)
blockcopy
      MOVS    r3,r2, LSR #3         ; Number of eight word multiples
      BEQ     copywords             ; Fewer than eight words to move?
      PUSH    {r4-r11}              ; Save some working registers
octcopy
      LDM     r0!, {r4-r11}         ; Load 8 words from the source
      STM     r1!, {r4-r11}         ; and put them at the destination
      SUBS    r3, r3, #1            ; Decrement the counter
      BNE     octcopy               ; ... copy more
      POP     {r4-r11}              ; Don't require these now - restore
                                    ; originals
copywords
      ANDS    r2, r2, #7            ; Number of odd words to copy
      BEQ     stop                  ; No words left to copy?
wordcopy
      LDR     r3, [r0], #4          ; Load a word from the source and
      STR     r3, [r1], #4          ; store it to the destination
      SUBS    r2, r2, #1            ; Decrement the counter
      BNE     wordcopy              ; ... copy more
stop
      MOV     r0, #0x18             ; angel_SWIreason_ReportException
      LDR     r1, =0x20026          ; ADP_Stopped_ApplicationExit

SVC   #0x123456             ; AArch32 semihosting (formerly SWI)

      AREA    BlockData, DATA, READWRITE
src   DCD     1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4
dst   DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      END
```


Note

The purpose of this example is to show the use of the LDM and STM instructions. There are other ways to perform bulk copy operations, the most efficient of which depends on many factors and is outside the scope of this document.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.3.19 Memory accesses

Many load and store instructions support different addressing modes.

Offset addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access. The base register is unchanged. The assembly language syntax for this mode is:

```
[<Rn>, <offset>]
```

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register. The assembly language syntax for this mode is:

```
[<Rn>, <offset>]!
```

Post-indexed addressing

The address obtained from the base register is used, unchanged, as the address for the memory access. The offset value is applied to the address, and written back into the base register. The assembly language syntax for this mode is:

```
[<Rn>], <offset>
```

In each case, `<Rn>` is the base register and `<offset>` can be:

- An immediate constant.
- An index register, `<Rm>`.
- A shifted index register, such as `<Rm>, LSL #<shift>`.

Related information

Address alignment in A32/T32 code on page 945

6.3.20 The Read-Modify-Write operation

The read-modify-write operation ensures that you modify only the specific bits in a system register that you want to change.

Individual bits in a system register control different system functionality. Modifying the wrong bits in a system register might cause your program to behave incorrectly.

```
    VMRS    r10,FPSCR             ; copy FPSCR into the general-purpose r10

    BIC     r10,r10,#0x00370000   ; clear STRIDE bits[21:20] and LEN bits[18:16]
    ORR     r10,r10,#0x00030000   ; set bits[17:16] (STRIDE =1 and LEN = 4)
    VMSR    FPSCR,r10             ; copy r10 back into FPSCR
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

To read-modify-write a system register, the instruction sequence is:

1. The first instruction copies the value from the target system register to a temporary general-purpose register.
2. The next one or more instructions modify the required bits in the general-purpose register. This can be one or both of:
   - `BIC` to clear to 0 only the bits that must be cleared.
   - `ORR` to set to 1 only the bits that must be set.
3. The final instruction writes the value from the general-purpose register to the target system register.

6.3.21 Optional hash with immediate constants

You do not have to specify a hash before an immediate constant in any instruction syntax.

This applies to A32, T32, Advanced SIMD, and floating-point instructions. For example, the following are valid instructions:

```
    BKPT 100
    MOVT R1, 256
    VCEQ.I8 Q1, Q2, 0
```

By default, the assembler warns if you do not specify a hash:

```
WARNING: A1865W: '#' not seen before constant expression.
```

You can suppressed this with `--diag_suppress=1865`.

If you use the assembly code with another assembler, you are advised to use the # before all immediates. The disassembler always shows the # for clarity.

6.3.22 Use of macros

A macro definition is a block of code enclosed between `MACRO` and `MEND` directives. It defines a name that you can use as a convenient alternative to repeating the block of code.

The main uses for a macro are:

- To make it easier to follow the logic of the source code by replacing a block of code with a single meaningful name.
- To avoid repeating a block of code several times.

Related information

Test-and-branch macro example on page 926
Unsigned integer division macro example on page 926

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

MACRO and MEND directives on page 1016

6.3.23 Test-and-branch macro example

You can use a macro to perform a test-and-branch operation.

In A32 code, a test-and-branch operation requires two instructions to implement.

You can define a macro such as this:

```
        MACRO

$label  TestAndBranch  $dest, $reg, $cc
$label  CMP     $reg, #0
        B$cc    $dest
        MEND
```

The line after the `MACRO` directive is the macro prototype statement. This defines the name (`TestAndBranch`) you use to invoke the macro. It also defines parameters (`$label`, `$dest`, `$reg`, and `$cc`). Unspecified parameters are substituted with an empty string. For this macro you must give values for `$dest`, `$reg` and `$cc` to avoid syntax errors. The assembler substitutes the values you give into the code.

This macro can be invoked as follows:

```
test    TestAndBranch    NonZero, r0, NE

          ...
          ...
NonZero
```

After substitution this becomes:

```
test    CMP     r0, #0

        BNE     NonZero
          ...
          ...
NonZero
```

Related information

Use of macros on page 925
Unsigned integer division macro example on page 926
Numeric local labels on page 953

6.3.24 Unsigned integer division macro example

You can use a macro to perform unsigned integer division.

The macro takes the following parameters:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

$Bot

The register that holds the divisor.

$Top

The register that holds the dividend before the instructions are executed. After the instructions are executed, it holds the remainder.

$Div

The register where the quotient of the division is placed. It can be `NULL` (`""`) if only the remainder is required.

$Temp

A temporary register used during the calculation.

Example unsigned integer division with a macro

```
        MACRO

$Lab    DivMod  $Div,$Top,$Bot,$Temp
        ASSERT  $Top <> $Bot        ; Produce an error message if the
        ASSERT  $Top <> $Temp       ; registers supplied are
        ASSERT  $Bot <> $Temp       ; not all different
        IF      "$Div" <> ""
            ASSERT  $Div <> $Top    ; These three only matter if $Div
            ASSERT  $Div <> $Bot    ; is not null ("")
            ASSERT  $Div <> $Temp   ;
        ENDIF
$Lab
        MOV     $Temp, $Bot           ; Put divisor in $Temp
        CMP     $Temp, $Top, LSR #1   ; double it until
90      MOVLS   $Temp, $Temp, LSL #1  ; 2 * $Temp > $Top
        CMP     $Temp, $Top, LSR #1
        BLS     %b90                  ; The b means search backwards
        IF      "$Div" <> ""          ; Omit next instruction if $Div
                                      ; is null
            MOV     $Div, #0          ; Initialize quotient
        ENDIF
91      CMP     $Top, $Temp           ; Can we subtract $Temp?
        SUBCS   $Top, $Top,$Temp      ; If we can, do so
        IF      "$Div" <> ""          ; Omit next instruction if $Div
                                      ; is null
            ADC     $Div, $Div, $Div  ; Double $Div
        ENDIF
        MOV     $Temp, $Temp, LSR #1  ; Halve $Temp,
        CMP     $Temp, $Bot           ; and loop until
        BHS     %b91                  ; less than divisor
        MEND
```

The macro checks that no two parameters use the same register. It also optimizes the code produced if only the remainder is required.

To avoid multiple definitions of labels if `DivMod` is used more than once in the assembler source, the macro uses numeric local labels (90, 91).

The following example shows the code that this macro produces if it is invoked as follows:

```
ratio  DivMod  R0,R5,R4,R2
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Output from the example division macro

```
        ASSERT  r5 <> r4          ; Produce an error if the

        ASSERT  r5 <> r2          ; registers supplied are
        ASSERT  r4 <> r2          ; not all different
        ASSERT  r0 <> r5          ; These three only matter if $Div
        ASSERT  r0 <> r4          ; is not null ("")
        ASSERT  r0 <> r2          ;
ratio
        MOV     r2, r4            ; Put divisor in $Temp
        CMP     r2, r5, LSR #1    ; double it until
90      MOVLS   r2, r2, LSL #1    ; 2 * r2 > r5
        CMP     r2, r5, LSR #1
        BLS     %b90              ; The b means search backwards
        MOV     r0, #0            ; Initialize quotient
91      CMP     r5, r2            ; Can we subtract r2?
        SUBCS   r5, r5, r2        ; If we can, do so
        ADC     r0, r0, r0        ; Double r0
        MOV     r2, r2, LSR #1    ; Halve r2,
        CMP     r2, r4            ; and loop until
        BHS     %b91              ; less than divisor
```

Related information

Use of macros on page 925
Test-and-branch macro example on page 926
Numeric local labels on page 953

6.3.25 Instruction and directive relocations

The assembler can embed relocation directives in object files to indicate labels with addresses that are unknown at assembly time. The assembler can relocate several types of instruction.

A relocation is a directive embedded in the object file that enables source code to refer to a label whose target address is unknown or cannot be calculated at assembly time. The assembler emits a relocation in the object file, and the linker resolves this to the address where the target is placed.

The assembler relocates the data directives `DCB` , `DCW`, `DCWU`, `DCD`, and `DCDU` if their syntax contains an external symbol, that is a symbol declared using `IMPORT` or `EXTERN`. This causes the bottom 8, 16, or 32 bits of the address to be used at link-time.

The `REQUIRE` directive emits a relocation to signal to the linker that the target label must be present if the current section is present.

The assembler is permitted to emit a relocation for these instructions:

`LDR` (PC-relative)

All A32 and T32 instructions, except the T32 doubleword instruction, can be relocated.

`PLD`, `PLDW`, and `PLI`

All A32 and T32 instructions can be relocated.

`B`, `BL`, and `BLX`

All A32 and T32 instructions can be relocated.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

CBZ and CBNZ

All T32 instructions can be relocated but this is discouraged because of the limited branch range of these instructions.

LDC and LDC2

Only A32 instructions can be relocated.

VLDR

Only A32 instructions can be relocated.

The assembler emits a relocation for these instructions if the label used meets any of the following requirements, as appropriate for the instruction type:

- The label is WEAK.
- The label is not in the same AREA.
- The label is external to the object (IMPORT or EXTERN).

For B, BL, and BX instructions, the assembler emits a relocation also if:

- The label is a function.
- The label is exported using EXPORT or GLOBAL.



Note

You can use the RELOC directive to control the relocation at a finer level, but this requires knowledge of the ABI.

Example

```
    IMPORT sym   ; sym is an external symbol

    DCW sym      ; Because DCW only outputs 16 bits, only the lower
                 ; 16 bits of the address of sym are inserted at
                 ; link-time.
```

Related information

AREA directive on page 974
EXPORT or GLOBAL directive on page 993
IMPORT and EXTERN directives on page 1010
REQUIRE directive on page 1024
RELOC directive on page 1023
DCB directive on page 984
DCD and DCDU directives on page 984
DCW and DCWU directives on page 990
ELF for the Arm Architecture

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.3.26 Symbol versions

The Arm linker conforms to the *Base Platform ABI for the Arm Architecture* (BPABI) and supports the GNU-extended symbol versioning model.

To add a symbol version to an existing symbol, you must define a version symbol at the same address. A version symbol is of the form:

- `<name>@<ver>` if `<ver>` is a non default version of `<name>`.
- `<name>@@<ver>` if `<ver>` is the default version of `<name>`.

The version symbols must be enclosed in vertical bars.

For example, to define a default version:

```
|my_versioned_symbol@@ver2|   ; Default version
my_asm_function    PROC
                   ...
                   BX lr
                   ENDP
```

To define a non default version:

```
|my_versioned_symbol@ver1|    ; Non default version
my_old_asm_function    PROC
                       ...
                       BX lr
                       ENDP
```

Related information

Accessing and Managing Symbols with armlink on page 621

6.3.27 Frame directives

Frame directives provide information in object files that enables debugging and profiling of assembly language functions.

You must use frame directives to describe the way that your code uses the stack if you want to be able to do either of the following:

- Debug your application using stack unwinding.
- Use either flat or call-graph profiling.

The assembler uses frame directives to insert DWARF debug frame information into the object file in ELF format that it produces. This information is required by a debugger for stack unwinding and for profiling.

Be aware of the following:

- Frame directives do not affect the code produced by the assembler.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The assembler does not validate the information in frame directives against the instructions emitted.

Related information

Exception tables and Unwind tables on page 931
About frame directives on page 968
Procedure Call Standard for the Arm Architecture

6.3.28 Exception tables and Unwind tables

You use `FRAME` directives to enable the assembler to generate unwind tables.


Note

Not supported for AArch64 state.

Exception tables are necessary to handle exceptions thrown by functions in high-level languages such as C++. Unwind tables contain debug frame information which is also necessary for the handling of such exceptions. An exception can only propagate through a function with an unwind table.

An assembly language function is code enclosed by either `PROC` and `ENDP` or `FUNC` and `ENDFUNC` directives. Functions written in C++ have unwind information by default. However, for assembly language functions that are called from C++ code, you must ensure that there are exception tables and unwind tables to enable the exceptions to propagate through them.

An exception cannot propagate through a function with a nounwind table. The exception handling runtime environment terminates the program if it encounters a nounwind table during exception processing.

The assembler can generate nounwind table entries for all functions and non-functions. The assembler can generate an unwind table for a function only if the function contains sufficient `FRAME` directives to describe the use of the stack within the function. To be able to create an unwind table for a function, each `POP` or `PUSH` instruction must be followed by a `FRAME POP` or `FRAME PUSH` directive respectively. Functions must conform to the conditions set out in the *Exception Handling ABI for the Arm Architecture* (EHABI), section 9.1 *Constraints on Use*. If the assembler cannot generate an unwind table it generates a nounwind table.

Related information

About frame directives on page 968
--exceptions, --no_exceptions (armasm) on page 877
--exceptions_unwind, --no_exceptions_unwind on page 878
FRAME UNWIND ON directive on page 1004
FRAME UNWIND OFF directive on page 1004
FUNCTION or PROC directive on page 1005

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

ENDFUNC or ENDP directive on page 991
Frame directives on page 930
Exception Handling ABI for the Arm Architecture

6.3.29 Extended notation extension for floating-point code

`armasm` implements an extension to the architectural floating-point assembly syntax, called extended notation. This extension allows you to include data type information or scalar indexes in register names.

If you use extended notation, you do not have to include the data type or scalar index information in every instruction.

Register names can be any of the following:

Untyped
The register name specifies the register, but not what data type it contains, nor any index to a particular scalar within the register.

Untyped with scalar index
The register name specifies the register, but not what data type it contains, It specifies an index to a particular scalar within the register.

Typed
The register name specifies the register, and what data type it contains, but not any index to a particular scalar within the register.

Typed with scalar index
The register name specifies the register, what data type it contains, and an index to a particular scalar within the register.

Use the `SN` and `DN` directives to define names for typed and scalar registers.

Related information
Arm Instruction Set for Armv7 and earlier Arm Architectures Reference Guide

6.3.30 Extended notation extension for Advanced SIMD

`armasm` implements an extension to the architectural Advanced SIMD assembly syntax, called extended notation. This extension allows you to include data type information or scalar indexes in register names.

If you use extended notation, you do not have to include the data type or scalar index information in every instruction.

Register names can be any of the following:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Untyped

The register name specifies the register, but not what data type it contains, nor any index to a particular scalar within the register.

Untyped with scalar index

The register name specifies the register, but not what data type it contains, It specifies an index to a particular scalar within the register.

Typed

The register name specifies the register, and what data type it contains, but not any index to a particular scalar within the register.

Typed with scalar index

The register name specifies the register, what data type it contains, and an index to a particular scalar within the register.

Use the `DN` and `QN` directives to define names for typed and scalar registers.

Related information

Arm Instruction Set for Armv7 and earlier Arm Architectures Reference Guide

6.3.31 Normal Advanced SIMD instructions

Many Advanced SIMD data processing instructions are available in Normal, Long, Wide, and Narrow variants.

`armasm` has no restrictions for the Long, Wide, and Narrow variants. By appending a `Q` to the instruction mnemonic for the Normal variant, the operands and result of a normal Advanced SIMD instruction are quadwords. However, if the operands or result are not quadwords, `armasm` produces an error. For example:

```
VADD.I16 Q0, Q1, Q2
```

Related information

Normal, long, wide, and narrow Advanced SIMD instructions

6.4 Using armasm

You can use a command line to invoke `armasm`. You must specify an input source file and you can specify various options to control the assembly.



Note

The `armasm` legacy assembler is deprecated, and it has not been updated since Arm® Compiler 6.10. Also, `armasm` does not support:

- Armv8.4-A or later architectures.
- Certain backported options in Armv8.2-A and Armv8.3-A.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Assembling SVE instructions.
- Armv8.1-M or later architectures, including MVE.
- All versions of the Armv8-R architecture.

As a reminder, `armasm` always reports the deprecation warning `A1950W`. To suppress this message, specify the `--diag_suppress=1950` option.

Syntax

```
armasm {<options>} <inputfile>
```

Parameters

`<options>`

Commands that instruct the assembler how to assemble the `<inputfile>`. You can invoke `armasm` with any combination of options separated by spaces. You can specify values for some options. To specify a value for an option, use either '=' (`<option>=<value>`) or a space character (`<option> <value>`).

`<inputfile>`

An assembly source file. It must contain UAL, pre-UAL A32 or T32, or A64 assembly language.

The assembler command line is case-insensitive, except in filenames and where specified. The assembler uses the same command-line ordering rules as the compiler. This ordering means that if the command line contains conflicting options, then the last option found always takes precedence.

6.4.1 Specify command-line options with an environment variable

The `ARMCOMPILER6_ASMOPT` environment variable can hold command-line options for the assembler.

The syntax is identical to the command-line syntax. The assembler reads the value of `ARMCOMPILER6_ASMOPT` and inserts it at the front of the command string. This means that options specified in `ARMCOMPILER6_ASMOPT` can be overridden by arguments on the command line.

Related information

Using armasm on page 933
Toolchain environment variables

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.4.2 Using stdin to input source code to the assembler

You can use stdin to pipe output from another program into armasm or 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.

About this task

To use `stdin` to pipe output from another program into `armasm`, invoke the program and the assembler using the pipe character (`|`). Use the minus character (`-`) as the source filename to instruct the assembler to take input from `stdin`. You must specify the output filename using the `-o` option. You can specify the command-line options you want to use. For example to pipe output from `fromelf`:

```
fromelf --disassemble A32input.o | armasm --cpu=8-A.32 -o A32output.o -
```



Note

The source code from `stdin` is stored in an internal cache that can hold up to 8 MB. You can increase this cache size using the `--maxcache` command-line option.

To use `stdin` to input source code directly on the command line:

Procedure

1. Invoke the assembler with the command-line options you want to use. Use the minus character (`-`) as the source filename to instruct the assembler to take input from `stdin`. You must specify the output filename using the `-o` option. For example:
   ```
   armasm --cpu=8-A.32 -o output.o -
   ```
2. Enter your input. For example:
   ```
           AREA     A32ex, CODE, READONLY

                                   ; Name this block of code A32ex
           ENTRY                   ; Mark first instruction to execute
   start
           MOV      r0, #10        ; Set up parameters
           MOV      r1, #3
           ADD      r0, r0, r1     ; r0 = r0 + r1
   stop
           MOV      r0, #0x18      ; angel_SWIreason_ReportException
           LDR      r1, =0x20026   ; ADP_Stopped_ApplicationExit
           SVC      #0x123456      ; AArch32 semihosting (formerly SWI)

           END                     ; Mark end of file
   ```
3. Terminate your input by entering:
   - `Ctrl+Z` then `Return` on Microsoft Windows systems.
   - `Ctrl+D` on Unix-based operating systems.

Related information

Using armasm on page 933
--maxcache=n on page 887

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.4.3 Built-in variables and constants

armasm defines built-in variables that hold information about, for example, the state of armasm, the command-line options used, and the target architecture or processor.

The following table lists the built-in variables defined by armasm:

Table 6-12: Built-in variables

{ARCHITECTURE}	Holds the name of the selected Arm® architecture.
{ARCHITECTURE}	Holds the name of the selected Arm architecture.
{AREANAME}	Holds the name of the current AREA.
{ARMASM_VERSION}	Holds an integer that increases with each version of armasm. The format of the version number is <M><mm><uu><xx> where: • <M> is the major version number, 6. • <mm> is the minor version number. • <uu> is the update number. • <xx> is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions. **Note:** The built-in variable |ads$version| is deprecated.
|ads$version|	Has the same value as {ARMASM_VERSION}.
{CODESIZE}	Is a synonym for {CONFIG}.
{COMMANDLINE}	Holds the contents of the command line.
{CONFIG}	Has the value: • 64 if the assembler is assembling A64 code. • 32 if the assembler is assembling A32 code. • 16 if the assembler is assembling T32 code.
{CPU}	Holds the name of the selected processor. The value of {CPU} is derived from the value specified in the --cpu option on the command line.
{ENDIAN}	Has the value "big" if the assembler is in big-endian mode, or "little" if it is in little-endian mode.
{FPU}	Holds the name of the selected FPU. The default in AArch32 state is "FP-ARMv8". The default in AArch64 state is "A64".
{INPUTFILE}	Holds the name of the current source file.
{INTER}	Has the Boolean value True if --apcs=/inter is set. The default is {False}.
{LINENUM}	Holds an integer indicating the line number in the current source file.
{LINENUMUP}	When used in a macro, holds an integer indicating the line number of the current macro. The value is the same as {LINENUM} when used in a non-macro context.
{LINENUMUPPER}	When used in a macro, holds an integer indicating the line number of the top macro. The value is the same as {LINENUM} when used in a non-macro context.
{OPT}	Value of the currently-set listing option. You can use the OPT directive to save the current listing option, force a change in it, or restore its original value.
{PC} or .	Address of current instruction.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

{ARCHITECTURE}	Holds the name of the selected Arm® architecture.
{PCSTOREOFFSET}	Is the offset between the address of the STR PC,[...] or STM Rb,{..., PC} instruction and the value of PC stored out. This varies depending on the processor or architecture specified.
{ROPI}	Has the Boolean value {True} if --apcs=/ropi is set. The default is {False}.
{RWPI}	Has the Boolean value {True} if --apcs=/rwpi is set. The default is {False}.
{VAR} or @	Current value of the storage area location counter.

You can use built-in variables in expressions or conditions in assembly source code. For example:

```
IF {ARCHITECTURE} = "8-A"
```

They cannot be set using the SETA, SETL, or SETS directives.

The names of the built-in variables can be in uppercase, lowercase, or mixed, for example:

```
IF {CpU} = "Generic ARM"
```

Note

All built-in string variables contain case-sensitive values. Relational operations on these built-in variables do not match with strings that contain an incorrect case. Use the command-line options --cpu and --fpu to determine valid values for {CPU}, {ARCHITECTURE}, and {FPU}.

The assembler defines the built-in Boolean constants TRUE and FALSE.

Table 6-13: Built-in Boolean constants

{FALSE}	Logical constant false.
{TRUE}	Logical constant true.

The following table lists the target processor-related built-in variables that are predefined by the assembler. Where the value field is empty, the symbol is a Boolean value and the meaning column describes when its value is {TRUE}.

Table 6-14: Predefined macros

Name	Value	Meaning
{TARGET_ARCH_AARCH32}	boolean	{TRUE} when assembling for AArch32 state. {FALSE} when assembling for AArch64 state.
{TARGET_ARCH_AARCH64}	boolean	{TRUE} when assembling for AArch64 state. {FALSE} when assembling for AArch32 state.
{TARGET_ARCH_ARM}	<num>	The number of the A32 base architecture of the target processor irrespective of whether the assembler is assembling for A32 or T32. The value is defined as zero when assembling for A64, and eight when assembling for A32/T32.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Name	Value	Meaning
{TARGET_ARCH_THUMB}	<num>	The number of the T32 base architecture of the target processor irrespective of whether the assembler is assembling for A32 or T32. The value is defined as zero when assembling for A64, and five when assembling for A32/T32.
{TARGET_ARCH_<XX>}	-	<XX> represents the target architecture and its value depends on the target processor: For the Armv8 architecture: • If you specify the assembler option --cpu=8-A.32 or --cpu=8-A.64 then {TARGET_ARCH_8_A} is defined. • If you specify the assembler option --cpu=8.1-A.32 or --cpu=8.1-A.64 then {TARGET_ARCH_8_1_A} is defined. For the Armv7 architecture, if you specify --cpu=Cortex-A8, for example, then {TARGET_ARCH_7_A} is defined.
{TARGET_FEATURE_EXTENSION_REGISTER_COUNT}	<num>	The number of 64-bit extension registers available in Advanced SIMD or floating-point.
{TARGET_FEATURE_CLZ}	-	If the target processor supports the CLZ instruction.
{TARGET_FEATURE_CRYPTOGRAPHY}	-	If the target processor has cryptographic instructions.
{TARGET_FEATURE_DIVIDE}	-	If the target processor supports the hardware divide instructions SDIV and UDIV.
{TARGET_FEATURE_DOUBLEWORD}	-	If the target processor supports doubleword load and store instructions, for example the A32 and T32 instructions LDRD and STRD (except the Armv6-M architecture).
{TARGET_FEATURE_DSPMUL}	-	If the DSP-enhanced multiplier (for example the SMLA<xy> instruction) is available.
{TARGET_FEATURE_MULTIPLY}	-	If the target processor supports long multiply instructions, for example the A32 and T32 instructions SMULL, SMLAL, UMULL, and UMLAL (that is, all architectures except the Armv6-M architecture).
{TARGET_FEATURE_MULTIPROCESSING}	-	If assembling for a target processor with Multiprocessing Extensions.
{TARGET_FEATURE_NEON}	-	If the target processor has Advanced SIMD.
{TARGET_FEATURE_NEON_FP16}	-	If the target processor has Advanced SIMD with half-precision floating-point operations.
{TARGET_FEATURE_NEON_FP32}	-	If the target processor has Advanced SIMD with single-precision floating-point operations.
{TARGET_FEATURE_NEON_INTEGER}	-	If the target processor has Advanced SIMD with integer operations.
{TARGET_FEATURE_UNALIGNED}	-	If the target processor has support for unaligned accesses (all architectures except the Armv6-M architecture).
{TARGET_FPU_SOFTVFP}	-	If assembling with the option --fpu=SoftVFP.
{TARGET_FPU_SOFTVFP_VFP}	-	If assembling for a target processor with SoftVFP and floating-point hardware, for example --fpu=SoftVFP+FP-ARMv8.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Name	Value	Meaning
{TARGET_FPU_VFP}	-	If assembling for a target processor with floating-point hardware, without using SoftVFP, for example --fpu=FP-ARMv8.
{TARGET_FPU_VFPV2}	-	If assembling for a target processor with VFPv2.
{TARGET_FPU_VFPV3}	-	If assembling for a target processor with VFPv3.
{TARGET_FPU_VFPV4}	-	If assembling for a target processor with VFPv4.
{TARGET_PROFILE_A}	-	If assembling for a Cortex®-A profile processor, for example, if you specify the assembler option --cpu=7-A.
{TARGET_PROFILE_M}	-	If assembling for a Cortex-M profile processor, for example, if you specify the assembler option --cpu=7-M.
{TARGET_PROFILE_R}	-	If assembling for a Cortex-R profile processor, for example, if you specify the assembler option --cpu=7-R.

Related information

--cpu=name (armasm) on page 867
--fpu=name (armasm) on page 882
Identifying versions of armasm in source code on page 939

6.4.4 Identifying versions of armasm in source code

The assembler defines the built-in variable ARMASM_VERSION to hold the version number of the assembler.

You can use it as follows:

```
IF ( {ARMASM_VERSION} / 100000) >= 6
    ; using armasm in Arm Compiler for Embedded 6
ELIF ( {ARMASM_VERSION} / 1000000) = 5
    ; using armasm in Arm Compiler 5
ELSE
    ; using armasm in Arm Compiler 4.1 or earlier
ENDIF
```



Note

The built-in variable |ads$version| is deprecated.

Related information

Built-in variables and constants on page 936

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.4.5 Diagnostic messages

The assembler can provide extra error, warning, and remark diagnostic messages in addition to the default ones.

By default, these additional diagnostic messages are not displayed. However, you can enable them using the command-line options `--diag_error`, `--diag_warning`, and `--diag_remark`.

Related information

--diag_error=tag[,tag,...] (armasm) on page 872
Interlocks diagnostics on page 940
Automatic IT block generation in T32 code on page 940
T32 branch target alignment on page 941
T32 code size diagnostics on page 941
A32 and T32 instruction portability diagnostics on page 942
T32 instruction width diagnostics on page 942
Two pass assembler diagnostics on page 943

6.4.6 Interlocks diagnostics

`armasm` can report warning messages about possible interlocks in your code caused by the pipeline of the processor chosen by the `--cpu` option.

To do this, use the `--diag_warning 1563` command-line option when invoking `armasm`.



Note

- `armasm` does not have an accurate model of the target processor, so these messages are not reliable when used with a multi-issue processor such as Cortex®-A8.
- Interlocks diagnostics apply to A32 and T32 code, but not to A64 code.

Related information

--diag_warning=tag[,tag,...] (armasm) on page 875
Automatic IT block generation in T32 code on page 940
T32 branch target alignment on page 941
T32 instruction width diagnostics on page 942
Diagnostic messages on page 939

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.4.7 Automatic IT block generation in T32 code

armasm can automatically insert an IT block for conditional instructions in T32 code, without requiring the use of explicit IT instructions.

If you write the following code:

```
        AREA x, CODE

        THUMB
        MOVNE   r0,r1
        NOP
        IT      NE
        MOVNE   r0,r1
        END
```

armasm generates the following instructions:

```
        IT      NE

        MOVNE   r0,r1
        NOP
        IT      NE
        MOVNE   r0,r1
```

You can receive warning messages about the automatic generation of IT blocks when assembling T32 code. To do this, use the armasm --diag_warning 1763 command-line option when invoking armasm.

Related information

Diagnostic messages on page 939
--diag_warning=tag[,tag,...] (armasm) on page 875

6.4.8 T32 branch target alignment

armasm can issue warnings about non word-aligned branch targets in T32 code.

On some processors, non word-aligned T32 instructions sometimes take one or more additional cycles to execute in loops. This means that it can be an advantage to ensure that branch targets are word-aligned. To ensure armasm reports such warnings, use the --diag_warning 1604 command-line option when invoking it.

Related information

Diagnostic messages on page 939
--diag_warning=tag[,tag,...] (armasm) on page 875

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.4.9 T32 code size diagnostics

In T32 code, some instructions, for example a branch or `LDR` (PC-relative), can be encoded as either a 32-bit or 16-bit instruction. `armasm` chooses the size of the instruction encoding.

`armasm` can issue a warning when it assembles a T32 instruction to a 32-bit encoding when it could have used a 16-bit encoding.

To enable this warning, use the `--diag_warning 1813` command-line option when invoking `armasm`.

Related information

Instruction width selection in T32 code on page 946
--diag_warning=tag[,tag,...] (armasm) on page 875
Diagnostic messages on page 939

6.4.10 A32 and T32 instruction portability diagnostics

`armasm` can issue warnings about instructions that cannot assemble to both A32 and T32 code.

There are a few UAL instructions that can assemble as either A32 code or T32 code, but not both. You can identify these instructions in the source code using the `--diag_warning 1812` command-line option when invoking `armasm`.

It warns for any instruction that cannot be assembled in the other instruction set. This is only a hint, and other factors, like relocation availability or target distance might affect the accuracy of the message.

Related information

--diag_warning=tag[,tag,...] (armasm) on page 875
Diagnostic messages on page 939

6.4.11 T32 instruction width diagnostics

`armasm` can issue a warning when it assembles a T32 instruction to a 32-bit encoding when it could have used a 16-bit encoding.

If you use the `.w` specifier, the instruction is encoded in 32 bits even if it could be encoded in 16 bits. You can use a diagnostic warning to detect when a branch instruction could have been encoded in 16 bits, but has been encoded in 32 bits. To do this, use the `--diag_warning 1607` command-line option when invoking `armasm`.



Note

This diagnostic does not produce a warning for relocated branch instructions, because the final address is not known. The linker might even insert a veneer, if the branch is out of range for a 32-bit instruction.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

Diagnostic messages on page 939
--diag_warning=tag[,tag,...] (armasm) on page 875

6.4.12 Two pass assembler diagnostics

`armasm` can issue a warning about code that might not be identical in both assembler passes.

`armasm` is a two pass assembler and the input code that the assembler reads must be identical in both passes. If a symbol is defined after the `:DEF:` test for that symbol, then the code read in pass one might be different from the code read in pass two. `armasm` can warn in this situation.

To do this, use the `--diag_warning 1907` command-line option when invoking `armasm`.

Example

The following example shows that the symbol `foo` is defined after the `:DEF: foo` test.

```
    AREA x,CODE

    [ :DEF: foo
    ]
foo MOV r3, r4
    END
```

Assembling this code with `--diag_warning 1907` generates the message:

```
Warning A1907W: Test for this symbol has been seen and might cause failure in the
 second pass.
```

Related information

--diag_warning=tag[,tag,...] (armasm) on page 875
Automatic IT block generation in T32 code on page 940
T32 branch target alignment on page 941
T32 instruction width diagnostics on page 942
Diagnostic messages on page 939
Directives that can be omitted in pass 2 of the assembler on page 969
How the assembler works

6.4.13 Using the C preprocessor

`armasm` can invoke `armclang` to preprocess an assembly language source file before assembling it. Preprocessing with `armclang` allows you to use C preprocessor commands in assembly source code.

If you require `armclang` preprocessing, you must use the `--cpreproc` command-line option together with the `--cpreproc_opts` command-line option when invoking the assembler. Including these options causes `armasm` to call `armclang` to preprocess the file before assembling it.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential



Note

As a minimum, you must specify the `armclang` option `--target` and either the `-mcpu` or `-march` option with `--cpreproc_opts`.

To assemble code containing C directives that require the C preprocessor, the input assembly source filename must have an upper-case extension `.s`. If your source filenames have a lower-case extension `.s`, then to avoid having to rename the files:

1. Perform the pre-processing step manually using the `armclang -x assembler-with-cpp` option.
2. Assemble the preprocessed file without using the `--cpreproc` and `--cpreproc_opts` options.

`armasm` looks for the `armclang` binary in the same directory as the `armasm` binary. If it does not find the binary, `armasm` expects the `armclang` binary to be on the PATH.

If present on the command line, `armasm` passes the following options by default to `armclang`:

- Basic pre-processor configuration options, such as `-E`.
- User-specified include directories, `-I` directives.
- Anything that is specified in `--cpreproc_opts`.

Some of the options that `armasm` passes to `armclang` are converted to the `armclang` equivalent beforehand. These options are shown in the following table:

Table 6-15: armclang equivalent command-line options

armasm	armclang
`--thumb`	`-mthumb`
`--arm`	`-marm`
`-i`	`-I`

`armasm` correctly interprets the preprocessed `#line` commands. It can generate error messages and `debug_line` tables using the information in the `#line` commands.

Preprocessing an assembly language source file

The following example shows the command that you write to preprocess and assemble a file, `source.s`. The example also passes the compiler options to define a macro that is called `RELEASE`, and to undefine a macro that is called `ALPHA`.

```
armasm --cpu=cortex-m3 --cpreproc --cpreproc_opts=--target=arm-arm-none-eabi,-
mcpu=cortex-a9,-D,RELEASE,-U,ALPHA source.S
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Preprocessing an assembly language source file manually

Alternatively, you must manually call `armclang` to preprocess the file before calling `armasm`. The following example shows the commands that you write to manually preprocess and assemble a file, `source.s`:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -E source.S > preprocessed.S
armasm --cpu=cortex-m3 preprocessed.S
```

In this example, the preprocessor outputs a file that is called `preprocessed.s`, and `armasm` assembles it.

Related information

--cpreproc on page 865
--cpreproc_opts=option[,option,...] on page 865
-march on page 123
-mcpu on page 150
--target on page 221
Specifying a target architecture, processor, and instruction set
Mandatory armclang options

6.4.14 Address alignment in A32/T32 code

In Arm®v7-A, Armv7-R, Armv8-A, and Armv8-R, the A bit in the System Control Register (SCTLR) controls whether alignment checking is enabled or disabled. In Armv7-M and Armv8-M, the `UNALIGN_TRP` bit, bit 3, in the Configuration and Control Register (CCR) controls the alignment checking.

If alignment checking is enabled, all unaligned word and halfword transfers cause an alignment exception. If disabled, unaligned accesses are permitted for the `LDR`, `LDRH`, `STR`, `STRH`, `LDRSH`, `LDRT`, `STRT`, `LDRSHT`, `LDRHT`, `STRHT`, and `TBH` instructions. Other data-accessing instructions always cause an alignment exception for unaligned data.

For `STRD` and `LDRD`, the specified address must be word-aligned.

If all your data accesses are aligned, you can use the `--no_unaligned_access` command-line option to declare that the output object was not permitted to make unaligned access. If all input objects declare that they are not permitted to use unaligned accesses, then the linker can avoid linking in any library functions that support unaligned access.

Related information

--unaligned_access, --no_unaligned_access (armasm) on page 894

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.4.15 Address alignment in A64 code

If alignment checking is not enabled, then unaligned accesses are permitted for all load and store instructions other than exclusive load, exclusive store, load acquire, and store release instructions. If alignment checking is enabled, then unaligned accesses are not permitted.

With alignment checking enabled, all load and store instructions must use addresses that are aligned to the size of the data being accessed:

- Addresses for 8-byte transfers must be 8-byte aligned.
- Addresses for 4-byte transfers are 4-byte word-aligned.
- Addresses for 2-byte transfers are 2-byte aligned.

Unaligned accesses cause an alignment exception.

For any memory access, if the stack pointer is used as the base register, then it must be quadword-aligned. Otherwise it generates a stack alignment exception.

If all your data accesses are aligned, you can use the `--no_unaligned_access` command-line option to declare that the output object was not permitted to make unaligned access. If all input objects declare that they are not permitted to use unaligned accesses, then the linker can avoid linking in any library functions that support unaligned access.

6.4.16 Instruction width selection in T32 code

Some T32 instructions can have either a 16-bit encoding or a 32-bit encoding.

If you do not specify the instruction size, by default:

- For forward reference `LDR`, `ADR`, and `B` instructions, `armasm` always generates a 16-bit instruction, even if that results in failure for a target that could be reached using a 32-bit instruction.
- For external reference `LDR` and `B` instructions, `armasm` always generates a 32-bit instruction.
- In all other cases, `armasm` generates the smallest size encoding that can be output.

If you want to override this behavior, you can use the `.W` or `.N` width specifier to ensure a particular instruction size. `armasm` faults if it cannot generate an instruction with the specified width.

The `.W` specifier is ignored when assembling to A32 code, so you can safely use this specifier in code that might assemble to either A32 or T32 code. However, the `.N` specifier is faulted when assembling to A32 code.

Related information

T32 code size diagnostics on page 941

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.5 Symbols, Literals, Expressions, and Operators in armasm Assembly Language

Describes how you can use symbols to represent variables, addresses, and constants in code, and how you can combine these with operators to create numeric or string expressions.

6.5.1 Symbol naming rules

You must follow some rules when naming symbols in assembly language source code.

The following rules apply:

- Symbol names must be unique within their scope.
- You can use uppercase letters, lowercase letters, numeric characters, or the underscore character in symbol names. Symbol names are case-sensitive, and all characters in the symbol name are significant.
- Do not use numeric characters for the first character of symbol names, except in numeric local labels.
- Symbols must not use the same name as built-in variable names or predefined symbol names.
- If you use the same name as an instruction mnemonic or directive, use double bars to delimit the symbol name. For example:

  ```
  ||ASSERT||
  ```

  The bars are not part of the symbol.
- You must not use the symbols `|$a|`, `|$t|`, or `|$d|` as program labels. These are mapping symbols that mark the beginning of A32, T32, and A64 code, and data within the object file. You must not use `|$x|` in A64 code.
- Symbols beginning with the characters $v are mapping symbols that relate to floating-point code. Arm recommends you avoid using symbols beginning with $v in your source code.

If you have to use a wider range of characters in symbols, for example, when working with compilers, use single bars to delimit the symbol name. For example:

```
|.text|
```

The bars are not part of the symbol. You cannot use bars, semicolons, or newlines within the bars.

Related information

Numeric local labels on page 953
Built-in variables and constants on page 936

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.5.2 Variables

You can declare numeric, logical, or string variables using assembler directives.

The value of a variable can be changed as assembly proceeds. Variables are local to the assembler. This means that in the generated code or data, every instance of the variable has a fixed value.

The type of a variable cannot be changed. Variables are one of the following types:

- Numeric.
- Logical.
- String.

The range of possible values of a numeric variable is the same as the range of possible values of a numeric constant or numeric expression.

The possible values of a logical variable are `<TRUE>` or `{FALSE}`.

The range of possible values of a string variable is the same as the range of values of a string expression.

Use the `GBLA`, `GBLL`, `GBLS`, `LCLA`, `LCLL`, and `LCLS` directives to declare symbols representing variables, and assign values to them using the `SETA`, `SETL`, and `SETS` directives.

Example

```
a    SETA 100

L1   MOV R1, #(a*5) ; In the object file, this is MOV R1, #500
a    SETA 200       ; Value of 'a' is 200 only after this point.
                    ; The previous instruction is always MOV R1, #500
     ...
     BNE L1         ; When the processor branches to L1, it executes
                    ; MOV R1, #500
```

Related information

Numeric expressions on page 956
String expressions on page 955
Numeric constants on page 948
Logical expressions on page 959
GBLA, GBLL, and GBLS directives on page 1006
LCLA, LCLL, and LCLS directives on page 1014
SETA, SETL, and SETS directives on page 1028

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.5.3 Numeric constants

You can define 32-bit numeric constants using the `EQU` assembler directive.

Numeric constants are 32-bit integers in A32 and T32 code. You can set them using unsigned numbers in the range 0 to 2^{32}-1, or signed numbers in therange -2^{31} to 2^{31}-1. However, the assembler makes no distinction between -<n> and 2^{32}-<n>.

In A64 code, numeric constants are 64-bit integers. You can set them using unsigned numbers in the range 0 to 2^{64}-1, or signed numbers in the range -2^{63} to 2^{63}-1. However, the assembler makes no distinction between -<n> and 2^{64}-<n>.

Relational operators such as >= use the unsigned interpretation. This means that 0 > -1 is `{FALSE}`.

Use the `EQU` directive to define constants. You cannot change the value of a numeric constant after you define it. You can construct expressions by combining numeric constants and binary operators.

Related information

Numeric expressions on page 956
Syntax of numeric literals on page 957
EQU directive on page 992

6.5.4 Assembly time substitution of variables

You can assign a string variable to all or part of a line of assembly language code. A string variable can contain numeric and logical variables.

Use the variable with a `$` prefix in the places where the value is to be substituted for the variable. The dollar character instructs `armasm` to substitute the string into the source code line before checking the syntax of the line. `armasm` faults if the substituted line is larger than the source line limit.

Numeric and logical variables can also be substituted. The current value of the variable is converted to a hexadecimal string (or `T` or `F` for logical variables) before substitution.

Use a dot to mark the end of the variable name if the following character would be permissible in a symbol name. You must set the contents of the variable before you can use it.

If you require a `$` that you do not want to be substituted, use `$$`. This is converted to a single `$`.

You can include a variable with a `$` prefix in a string. Substitution occurs in the same way as anywhere else.

Substitution does not occur within vertical bars, except that vertical bars within double quotes do not affect substitution.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

```
    ; straightforward substitution

            GBLS    add4ff
            ;
add4ff      SETS    "ADD  r4,r4,#0xFF"    ; set up add4ff
            $add4ff.00                    ; invoke add4ff
            ; this produces
            ADD  r4,r4,#0xFF00
    ; elaborate substitution
            GBLS    s1
            GBLS    s2
            GBLS    fixup
            GBLA    count
            ;
count       SETA    14
s1          SETS    "a$$b$count" ; s1 now has value a$b0000000E
s2          SETS    "abc"
fixup       SETS    "|xy$s2.z|"  ; fixup now has value |xyabcz|
|C$$code|   MOV     r4,#16       ; but the label here is C$$code
```

Related information

Syntax of source lines in armasm syntax assembly language on page 899
Symbol naming rules on page 947

6.5.5 Register-relative and PC-relative expressions

The assembler supports PC-relative and register-relative expressions.

A register-relative expression evaluates to a named register combined with a numeric expression.

You write a PC-relative expression in source code as a label or the PC, optionally combined with a numeric expression. Some instructions can also accept PC-relative expressions in the form `[PC, #number]`.

If you specify a label, the assembler calculates the offset from the PC value of the current instruction to the address of the label. The assembler encodes the offset in the instruction. If the offset is too large, the assembler produces an error. The offset is either added to or subtracted from the PC value to form the required address.

Arm recommends you write PC-relative expressions using labels rather than the PC because the value of the PC depends on the instruction set.


Note

- In A32 code, the value of the PC is the address of the current instruction plus 8 bytes.
- In T32 code:
  - For `B`, `BL`, `CBNZ`, and `CBZ` instructions, the value of the PC is the address of the current instruction plus 4 bytes.
  - For all other instructions that use labels, the value of the PC is the address of the current instruction plus 4 bytes, with bit[1] of the result cleared to 0 to make it word-aligned.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- In A64 code, the value of the PC is the address of the current instruction.

Example

```
        LDR     r4,=data+4*n    ; n is an assembly-time variable

        ; code
        MOV     pc,lr
data    DCD     value_0
        ; n-1 DCD directives
        DCD     value_n         ; data+4*n points here
        ; more DCD directives
```

Related information

Labels on page 951
MAP directive on page 1019

6.5.6 Labels

A label is a symbol that represents the memory address of an instruction or data.

The address can be PC-relative, register-relative, or absolute. Labels are local to the source file unless you make them global using the `EXPORT` directive.

The address given by a label is calculated during assembly. `armasm` calculates the address of a label relative to the origin of the section where the label is defined. A reference to a label within the same section can use the PC plus or minus an offset. This is called PC-relative addressing.

Addresses of labels in other sections are calculated at link time, when the linker has allocated specific locations in memory for each section.

Related information

Syntax of source lines in armasm syntax assembly language on page 899
EXPORT or GLOBAL directive on page 993
Labels for PC-relative addresses on page 951
Labels for register-relative addresses on page 952
Labels for absolute addresses on page 952

6.5.7 Labels for PC-relative addresses

A label can represent the PC value plus or minus the offset from the PC to the label. Use these labels as targets for branch instructions, or to access small items of data embedded in code sections.

You can define PC-relative labels using a label on an instruction or on one of the data definition directives.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

You can also use the section name of an AREA directive as a label for PC-relative addresses. In this case the label points to the first byte of the specified AREA. Arm does not recommend using AREA names as branch targets because when branching from A32 to T32 state or T32 to A32 state in this way, the processor does not change the state properly.

Related information

AREA directive on page 974
DCB directive on page 984
DCD and DCDU directives on page 984
DCFD and DCFDU directives on page 986
DCFS and DCFSU directives on page 987
DCI directive on page 987
DCQ and DCQU directives on page 989
DCW and DCWU directives on page 990

6.5.8 Labels for register-relative addresses

A label can represent a named register plus a numeric value. You define these labels in a storage map. They are most commonly used to access data in data sections.

You can use the EQU directive to define additional register-relative labels, based on labels defined in storage maps.



Note

Register-relative addresses are not supported in A64 code.

Example of storage map definitions

```
    MAP     0,r9
    MAP     0xff,r9
```

Related information

DCDO directive on page 985
EQU directive on page 992
MAP directive on page 1019
SPACE and FILL directives on page 1029

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.5.9 Labels for absolute addresses

A label can represent the absolute address of code or data.

These labels are numeric constants. In A32 and T32 code they are integers in the range 0 to 2^{32}-1. In A64 code, they are integers in the range 0 to 2^{64}-1. They address the memory directly. You can use labels to represent absolute addresses using the `EQU` directive. To ensure that the labels are used correctly when referenced in code, you can specify the absolute address as:

- A32 code with the `ARM` directive.
- T32 code with the `THUMB` directive.
- Data.

Example of defining labels for absolute address

```
abc EQU 2               ; assigns the value 2 to the symbol abc

xyz EQU label+8         ; assigns the address (label+8) to the symbol xyz
fiq EQU 0x1C, ARM       ; assigns the absolute address 0x1C to the symbol fiq
                        ; and marks it as A32 code
```

Related information

Labels on page 951
EQU directive on page 992
Labels for PC-relative addresses on page 951
Labels for register-relative addresses on page 952

6.5.10 Numeric local labels

Numeric local labels are a type of label that you refer to by number rather than by name. They are used in a similar way to PC-relative labels, but their scope is more limited.

A numeric local label is a number in the range 0-99, optionally followed by a name. Unlike other labels, a numeric local label can be defined many times and the same number can be used for more than one numeric local label in an area.

Numeric local labels do not appear in the object file. This means that, for example, a debugger cannot set a breakpoint directly on a numeric local label, like it can for named local labels kept using the `KEEP` directive.

A numeric local label can be used in place of `<symbol>` in source lines in an assembly language module:

- On its own, that is, where there is no instruction or directive.
- On a line that contains an instruction.
- On a line that contains a code- or data-generating directive.

A numeric local label is generally used where you might use a PC-relative label.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Numeric local labels are typically used for loops and conditional code within a routine, or for small subroutines that are only used locally. They are particularly useful when you are generating labels in macros.

The scope of numeric local labels is limited by the AREA directive. Use the ROUT directive to limit the scope of numeric local labels more tightly. A reference to a numeric local label refers to a matching label within the same scope. If there is no matching label within the scope in either direction, armasm generates an error message and the assembly fails.

You can use the same number for more than one numeric local label even within the same scope. By default, armasm links a numeric local label reference to:

- The most recent numeric local label with the same number, if there is one within the scope.
- The next following numeric local label with the same number, if there is not a preceding one within the scope.

Use the optional parameters to modify this search pattern if required.

Related information

Syntax of source lines in armasm syntax assembly language on page 899
Syntax of numeric local labels on page 954
Labels on page 951
MACRO and MEND directives on page 1016
KEEP directive on page 1013
ROUT directive on page 1027

6.5.11 Syntax of numeric local labels

When referring to numeric local labels you can specify how armasm searches for the label.

Syntax

```
<n> [<routname>] ; a numeric local label

% [F|B][A|T]<n>[<routname>]; a reference to a numeric local label
```

where:

<n>
is the number of the numeric local label in the range 0-99.

<routname>
is the name of the current scope.

%
introduces the reference.

F
instructs armasm to search forwards only.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

B
instructs `armasm` to search backwards only.

A
instructs `armasm` to search all macro levels.

T
instructs `armasm` to look at this macro level only.

Usage

If neither `F` nor `B` is specified, `armasm` searches backwards first, then forwards.

If neither `A` nor `T` is specified, `armasm` searches all macros from the current level to the top level, but does not search lower level macros.

If `<routname>` is specified in either a label or a reference to a label, `armasm` checks it against the name of the nearest preceding `ROUT` directive. If it does not match, `armasm` generates an error message and the assembly fails.

Related information

Numeric local labels on page 953
ROUT directive on page 1027

6.5.12 String expressions

String expressions consist of combinations of string literals, string variables, string manipulation operators, and parentheses.

Characters that cannot be placed in string literals can be placed in string expressions using the `:CHR:` unary operator. Any ASCII character from 0 to 255 is permitted.

The value of a string expression cannot exceed 5120 characters in length. It can be of zero length.

Example

```
improb  SETS    "literal":CC:(strvar2:LEFT:4)

            ; sets the variable improb to the value "literal"
            ; with the left-most four characters of the
            ; contents of string variable strvar2 appended
```

Related information

String literals on page 955
Unary operators on page 960
String manipulation operators on page 962
Variables on page 947
SETA, SETL, and SETS directives on page 1028

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.5.13 String literals

String literals consist of a series of characters or spaces contained between double quote characters.

The length of a string literal is restricted by the length of the input line.

To include a double quote character or a dollar character within the string literal, include the character twice as a pair. For example, you must use `$$` if you require a single `$` in the string.

C string escape sequences are also enabled and can be used within the string, unless `--no_esc` is specified.

Examples

```
abc     SETS    "this string contains only one "" double quote"
def     SETS    "this string contains only one $$ dollar symbol"
```

Related information

Syntax of source lines in armasm syntax assembly language on page 899
--no_esc on page 888

6.5.14 Numeric expressions

Numeric expressions consist of combinations of numeric constants, numeric variables, ordinary numeric literals, binary operators, and parentheses.

Numeric expressions can contain register-relative or program-relative expressions if the overall expression evaluates to a value that does not include a register or the PC.

Numeric expressions evaluate to 32-bit integers in A32 and T32 code. You can interpret them as unsigned numbers in the range 0 to 2^{32}-1, or signed numbers in the range -2^{31} to 2^{31}-1. However, `armasm` makes no distinction between -<n> and 2^{32}<n>. Relational operators such as >= use the unsigned interpretation. This means that 0 > -1 is `{FALSE}`.

In A64 code, numeric expressions evaluate to 64-bit integers. You can interpret them as unsigned numbers in the range 0 to 2^{64}-1, or signed numbers in the range -2^{63} to 2^{63}-1. However, `armasm` makes no distinction between -<n> and 2^{64}-<n>.

Note

`armasm` does not support 64-bit arithmetic variables. See SETA, SETL, and SETS directives (Restrictions) for a workaround.

Arm recommends that you only use `armasm` for legacy Arm® syntax assembly code, and that you use the `armclang` assembler and GNU syntax for all new assembly files.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

```
a   SETA    256*256       ; 256*256 is a numeric expression
    MOV     r1,#(a*22)    ; (a*22) is a numeric expression
```

Related information

Syntax of numeric literals on page 957
Binary operators on page 961
Variables on page 947
Numeric constants on page 948
SETA, SETL, and SETS directives on page 1028

6.5.15 Syntax of numeric literals

Numeric literals consist of a sequence of characters, or a single character in quotes, evaluating to an integer.

They can take any of the following forms:

- `<decimal-digits>.`
- `0x<hexadecimal-digits>.`
- `&<hexadecimal-digits>.`
- `<n_base-n-digits>.`
- `'<character>'.`

where:

`<decimal-digits>`
Is a sequence of characters using only the digits 0 to 9.

`<hexadecimal-digits>`
Is a sequence of characters using only the digits 0 to 9 and the letters A to F or a to f.

`<n_>`
Is a single digit between 2 and 9 inclusive, followed by an underscore character.

`<base-n-digits>`
Is a sequence of characters using only the digits 0 to (`<n>`-1)

`<character>`
Is any single character except a single quote. Use the standard C escape character (') if you require a single quote. The character must be enclosed within opening and closing single quotes. In this case, the value of the numeric literal is the numeric code of the character.

You must not use any other characters. The sequence of characters must evaluate to an integer.

In A32/T32 code, the range is 0 to 2^{32}-1, except in `DCQ` , `DCQU`, `DCD`, and `DCDU` directives.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

In A64 code, the range is 0 to 2^{64}-1, except in DCD and DCDU directives.


Note

- In the DCQ and DCQU, the integer range is 0 to 2^{64}-1
- In the DCO and DCOU directives, the integer range is 0 to 2^{128}-1

Examples

```
a       SETA    34906

addr    DCD     0xA10E
        LDR     r4,=&1000000F
        DCD     2_11001010
c3      SETA    8_74007
        DCQ     0x0123456789abcdef
        LDR     r1,='A'      ; pseudo-instruction loading 65 into r1
        ADD     r3,r2,#'\''  ; add 39 to contents of r2, result to r3
```

Related information

Numeric constants on page 948

6.5.16 Syntax of floating-point literals

Floating-point literals consist of a sequence of characters evaluating to a floating-point number.

They can take any of the following forms:

- `{-}<digits>E{-}<digits>`
- `{-}{\<digits>\}.<digits>`
- `{-}{\<digits>\}.<digits>E{-}<digits>`
- `0x<hexdigits>`
- `&<hexdigits>`
- `0f_<hexdigits>`
- `0d_<hexdigits>`

where:

`<digits>`

Sequences of characters using only the digits 0 to 9. You can write E in uppercase or lowercase. These forms correspond to normal floating-point notation.

`<hexdigits>`

Sequences of characters using only the digits 0 to 9 and the letters A to F or a to f. These forms correspond to the internal representation of the numbers in the computer. Use these forms to enter infinities and NaNs, or if you want to be sure of the exact bit patterns you are using.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The `0x` and `&` forms allow the floating-point bit pattern to be specified by any number of hex digits.

The `0f_` form requires the floating-point bit pattern to be specified by exactly 8 hex digits.

The `0d_` form requires the floating-point bit pattern to be specified by exactly 16 hex digits.

The range for half-precision floating-point values is:

- Maximum 65504 (IEEE format) or 131008 (alternative format).
- Minimum 0.000122010707855224 61.

The range for single-precision floating-point values is:

- Maximum 3.40282347e+38.
- Minimum 1.17549435e-38.

The range for double-precision floating-point values is:

- Maximum 1.79769313486231571e+308.
- Minimum 2.22507385850720138e-308.

Floating-point numbers are only available if your system has floating-point, Advanced SIMD with floating-point.

Examples

```
    DCFD    1E308,-4E-100
    DCFS    1.0
    DCFS    0.02
    DCFD    3.725e15
    DCFS    0x7FC00000              ; Quiet NaN
    DCFD    &FFF0000000000000       ; Minus infinity
```

Related information

Numeric constants on page 948
Syntax of numeric literals on page 957

6.5.17 Logical expressions

Logical expressions consist of combinations of logical literals (`{TRUE}` or `{FALSE}`), logical variables, Boolean operators, relations, and parentheses.

Relations consist of combinations of variables, literals, constants, or expressions with appropriate relational operators.

Related information

Boolean operators on page 964
Relational operators on page 963

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.5.18 Logical literals

Logical or Boolean literals can have one of two values, `{TRUE}` or `{FALSE}`.

Related information

Syntax of numeric literals on page 957
String literals on page 955

6.5.19 Unary operators

Unary operators return a string, numeric, or logical value. They have higher precedence than other operators and are evaluated first.

A unary operator precedes its operand. Adjacent operators are evaluated from right to left.

The following table lists the unary operators that return strings:

Table 6-16: Unary operators that return strings

Operator	Usage	Description
`:CHR:`	`:CHR:A`	Returns the character with ASCII code A.
`:LOWERCASE:`	`:LOWERCASE:string`	Returns the given string, with all uppercase characters converted to lowercase.
`:REVERSE_CC:`	`:REVERSE_CC:cond_code`	Returns the inverse of the condition code in `cond_code`, or an error if `cond_code` does not contain a valid condition code.
`:STR:`	`:STR:A`	In A32 and T32 code, returns an 8-digit hexadecimal string corresponding to a numeric expression, or the string `"T"` or `"F"` if used on a logical expression. In A64 code, returns a 16-digit hexadecimal string.
`:UPPERCASE:`	`:UPPERCASE:string`	Returns the given string, with all lowercase characters converted to uppercase.

The following table lists the unary operators that return numeric values:

Table 6-17: Unary operators that return numeric or logical values

Operator	Usage	Description
?	`?A`	Number of bytes of code generated by line defining symbol A.
+ and –	`+A` `-A`	Unary plus. Unary minus. + and - can act on numeric and PC-relative expressions.
`:BASE:`	`:BASE:A`	If A is a PC-relative or register-relative expression, `:BASE:` returns the number of its register component. `:BASE:` is most useful in macros.
`:CC_ENCODING:`	`:CC_ENCODING:cond_code`	Returns the numeric value of the condition code in `cond_code`, or an error if `cond_code` does not contain a valid condition code.
`:DEF:`	`:DEF:A`	{`TRUE`} if A is defined, otherwise {`FALSE`}.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Operator	Usage	Description
`:INDEX:`	`:INDEX:A`	If A is a register-relative expression, `:INDEX:` returns the offset from that base register. `:INDEX:` is most useful in macros.
`:LEN:`	`:LEN:A`	Length of string A.
`:LNOT:`	`:LNOT:A`	Logical complement of A.
`:NOT:`	`:NOT:A`	Bitwise complement of A (~ is an alias, for example `~A`).
`:RCONST:`	`:RCONST:Rn`	Number of register. In A32/T32 code, 0-15 corresponds to R0-R15. In A64 code, 0-30 corresponds to W0-W30 or X0-X30.

Related information

Binary operators on page 961

6.5.20 Binary operators

You write binary operators between the pair of sub-expressions they operate on. They have lower precedence than unary operators.



Note

The order of precedence is not the same as in C.

Related information

Multiplicative operators on page 961
String manipulation operators on page 962
Shift operators on page 962
Addition, subtraction, and logical operators on page 963
Relational operators on page 963
Boolean operators on page 964
Difference between operator precedence in assembly language and C on page 965

6.5.21 Multiplicative operators

Multiplicative operators have the highest precedence of all binary operators. They act only on numeric expressions.

The following table shows the multiplicative operators:

Table 6-18: Multiplicative operators

Operator	Alias	Usage	Explanation
*	No alias	`A*B`	Multiply
/	No alias	`A/B`	Divide
`:MOD:`	%	`A:MOD:B`	A modulo B

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

You can use the `:MOD:` operator on PC-relative expressions to ensure code is aligned correctly. These alignment checks have the form `<PC-relative>:MOD:<Constant>`. For example:

```
    AREA x,CODE

    ASSERT ({PC}:MOD:4) == 0
    DCB 1
y   DCB 2
    ASSERT (y:MOD:4) == 1
    ASSERT ({PC}:MOD:4) == 2
    END
```

Related information

Binary operators on page 961
Register-relative and PC-relative expressions on page 950
Syntax of numeric literals on page 957
Numeric expressions on page 956

6.5.22 String manipulation operators

You can use string manipulation operators to concatenate two strings, or to extract a substring.

The following table shows the string manipulation operators. In `:CC:`, both `A` and `B` must be strings. In the slicing operators `:LEFT:` and `:RIGHT:`:

- `A` must be a string.
- `B` must be a numeric expression.

Table 6-19: String manipulation operators

Operator	Usage	Explanation
`:CC:`	`A:CC:B`	B concatenated onto the end of A
`:LEFT:`	`A:LEFT:B`	The left-most B characters of A
`:RIGHT:`	`A:RIGHT:B`	The right-most B characters of A

Related information

String expressions on page 955
Numeric expressions on page 956

6.5.23 Shift operators

Shift operators act on numeric expressions, by shifting or rotating the first operand by the amount specified by the second.

The following table shows the shift operators:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 6-20: Shift operators

Operator	Alias	Usage	Explanation
`:ROL:`	No alias	`A:ROL:B`	Rotate A left by B bits
`:ROR:`	No alias	`A:ROR:B`	Rotate A right by B bits
`:SHL:`	<<	`A:SHL:B`	Shift A left by B bits
`:SHR:`	>>	`A:SHR:B`	Shift A right by B bits


Note

`SHR` is a logical shift and does not propagate the sign bit.

Related information

Binary operators on page 961

6.5.24 Addition, subtraction, and logical operators

Addition, subtraction, and logical operators act on numeric expressions.

Logical operations are performed bitwise, that is, independently on each bit of the operands to produce the result.

The following table shows the addition, subtraction, and logical operators:

Table 6-21: Addition, subtraction, and logical operators

Operator	Alias	Usage	Explanation
+	No alias	`A+B`	Add A to B
–	No alias	`A-B`	Subtract B from A
`:AND:`	`&`	`A:AND:B`	Bitwise AND of A and B
`:EOR:`	^	`A:EOR:B`	Bitwise Exclusive OR of A and B
`:OR:`	No alias	`A:OR:B`	Bitwise OR of A and B

The use of `|` as an alias for `:OR:` is deprecated.

Related information

Binary operators on page 961

6.5.25 Relational operators

Relational operators act on two operands of the same type to produce a logical value.

The operands can be one of:

- Numeric.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- PC-relative.
- Register-relative.
- Strings.

Strings are sorted using ASCII ordering. String `A` is less than string `B` if it is a leading substring of string `B`, or if the left-most character in which the two strings differ is less in string `A` than in string `B`.

Arithmetic values are unsigned, so the value of `0>-1` is `{FALSE}`.

The following table shows the relational operators:

Table 6-22: Relational operators

Operator	Alias	Usage	Explanation
=	==	A=B	A equal to B
>	No alias	A>B	A greater than B
>=	No alias	A>=B	A greater than or equal to B
<	No alias	A<B	A less than B
<=	No alias	A<=B	A less than or equal to B
/=	<> !=	A/=B	A not equal to B

Related information

Binary operators on page 961

6.5.26 Boolean operators

Boolean operators perform standard logical operations on their operands. They have the lowest precedence of all operators.

In all three cases, both A and B must be expressions that evaluate to either `{TRUE}` or `{FALSE}`.

The following table shows the Boolean operators:

Table 6-23: Boolean operators

Operator	Alias	Usage	Explanation
:LAND:	&&	A:LAND:B	Logical AND of A and B
:LEOR:	No alias	A:LEOR:B	Logical Exclusive OR of A and B
:LOR:	||	A:LOR:B	Logical OR of A and B

Related information

Binary operators on page 961

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.5.27 Operator precedence

`armasm` includes an extensive set of operators for use in expressions. It evaluates them using a strict order of precedence.

Many of the operators resemble their counterparts in high-level languages such as C.

`armasm` evaluates operators in the following order:

1. Expressions in parentheses are evaluated first.
2. Operators are applied in precedence order.
3. Adjacent unary operators are evaluated from right to left.
4. Binary operators of equal precedence are evaluated from left to right.

Related information

Unary operators on page 960
Binary operators on page 961
Multiplicative operators on page 961
String manipulation operators on page 962
Shift operators on page 962
Addition, subtraction, and logical operators on page 963
Relational operators on page 963
Boolean operators on page 964
Difference between operator precedence in assembly language and C on page 965

6.5.28 Difference between operator precedence in assembly language and C

`armasm` does not follow exactly the same order of precedence when evaluating operators as a C compiler.

For example, `(1 + 2 :SHR: 3)` evaluates as `(1 + (2 :SHR: 3))` = 1 in assembly language. The equivalent expression in C evaluates as `((1 + 2) >> 3)` = 0.

Arm recommends you use brackets to make the precedence explicit.

If your code contains an expression that would parse differently in C, and you are not using the `--unsafe` option, `armasm` gives a warning:

```
A1466W: Operator precedence means that expression would evaluate differently in C
```

In the following tables:

- The highest precedence operators are at the top of the list.
- The highest precedence operators are evaluated first.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Operators of equal precedence are evaluated from left to right.

The following table shows the order of precedence of operators in assembly language, and a comparison with the order in C.

Table 6-24: Operator precedence in Arm assembly language

Assembly language precedence	Equivalent C operators
unary operators	unary operators
`* / :MOD:`	`* / %`
string manipulation	n/a
`:SHL: :SHR: :ROR: :ROL:`	`<< >>`
`+ - :AND: :OR: :EOR:`	`+ - & | ^`
`= > >= < <= /= <>`	`== > >= < <= !=`
`:LAND: :LOR: :LEOR:`	`&& ||`

The order of precedence of operators in C is as follows:

1. C precedence
2. unary operators
3. `* / %`
4. `+` `-` (as binary operators)
5. `<< >>`
6. `< <= > >=`
7. `== !=`
8. `&`
9. `^`
10. `|`
11. `&&`
12. `||`

Related information

Operator precedence on page 964
Binary operators on page 961

6.6 armasm Directives Reference

Describes the directives that are provided by the Arm assembler, `armasm`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.1 Alphabetical list of directives armasm assembly language directives

The Arm assembler, armasm, provides various directives.

The following table lists them:

Table 6-25: List of directives

Directive	Directive	Directive
ALIAS	EQU	LTORG
ALIGN	EXPORT or GLOBAL	MACRO and MEND
ARM or CODE32	EXPORTAS	MAP
AREA	EXTERN	MEND (see MACRO)
ASSERT	FIELD	MEXIT
ATTR	FRAME ADDRESS	NOFP
CN	FRAME POP	OPT
CODE16	FRAME PUSH	PRESERVE8 (see REQUIRE8)
COMMON	FRAME REGISTER	PROC see FUNCTION
CP	FRAME RESTORE	-
DATA	FRAME SAVE	RELOC
DCB	FRAME STATE REMEMBER	REQUIRE
DCD and DCDU	FRAME STATE RESTORE	REQUIRE8 and PRESERVE8
DCDO	FRAME UNWIND ON or OFF	RLIST
DCFD and DCFDU	FUNCTION or PROC	RN
DCFS and DCFSU	GBLA, GBLL, and GBLS	ROUT
DCI	GET or INCLUDE	SETA, SETL, and SETS
DCQ and DCQU	GLOBAL (see EXPORT)	SN
DCW and DCWU	IF, ELSE, ENDIF, and ELIF	SPACE or FILL
DN	IMPORT	SUBT
ELIF, ELSE (see IF)	INCBIN	THUMB
END	INCLUDE see GET	TTL
ENDFUNC or ENDP	INFO	WHILE and WEND
ENDIF (see IF)	KEEP	WN and XN
ENTRY	LCLA, LCLL, and LCLS	-

6.6.2 About armasm assembly language control directives

Some armasm assembler directives control conditional assembly, looping, inclusions, and macros.

These directives are as follows:

- MACRO and MEND.
- MEXIT.
- IF, ELSE, ENDIF, and ELIF.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- `WHILE` and `WEND`.

Nesting directives

The following structures can be nested to a total depth of 256:

- `MACRO` definitions.
- `WHILE...WEND` loops.
- `IF...ELSE...ENDIF` conditional structures.
- `INCLUDE` file inclusions.

The limit applies to all structures taken together, regardless of how they are nested. The limit is not 256 of each type of structure.

Related information

MACRO and MEND directives on page 1016
MEXIT directive on page 1020
IF, ELSE, ENDIF, and ELIF directives on page 1008
WHILE and WEND directives on page 1032

6.6.3 About frame directives

Frame directives enable debugging and profiling of assembly language functions. They also enable the stack usage of functions to be calculated.

Correct use of these directives:

- Enables the `armlink --callgraph` option to calculate stack usage of assembler functions.

  The following are the rules that determine stack usage:

  - If a function is not marked with `PROC` or `ENDP`, stack usage is unknown.
  - If a function is marked with `PROC` or `ENDP` but with no `FRAME PUSH` or `FRAME POP`, stack usage is assumed to be zero. This means that there is no requirement to manually add `FRAME PUSH 0` or `FRAME POP 0`.
  - If a function is marked with `PROC` or `ENDP` and with `FRAME PUSH n` or `FRAME POP n`, stack usage is assumed to be `n` bytes.
- Helps you to avoid errors in function construction, particularly when you are modifying existing code.
- Enables the assembler to alert you to errors in function construction.
- Enables backtracing of function calls during debugging.
- Enables the debugger to profile assembler functions.

If you require profiling of assembler functions, but do not want frame description directives for other purposes:

- You must use the `FUNCTION` and `ENDFUNC`, or `PROC` and `ENDP`, directives.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- You can omit the other `FRAME` directives.
- You only have to use the `FUNCTION` and `ENDFUNC` directives for the functions you want to profile.

In DWARF, the canonical frame address is an address on the stack specifying where the call frame of an interrupted function is located.

Related information

FRAME ADDRESS directive on page 997
FRAME POP directive on page 998
FRAME PUSH directive on page 999
FRAME REGISTER directive on page 1000
FRAME RESTORE directive on page 1001
FRAME RETURN ADDRESS directive on page 1001
FRAME SAVE directive on page 1002
FRAME STATE REMEMBER directive on page 1003
FRAME STATE RESTORE directive on page 1003
FRAME UNWIND ON directive on page 1004
FRAME UNWIND OFF directive on page 1004
FUNCTION or PROC directive on page 1005
ENDFUNC or ENDP directive on page 991

6.6.4 Directives that can be omitted in pass 2 of the assembler

Most directives must appear in both passes of the assembly process. You can omit some directives from the second pass over the source code by the assembler, but doing this is strongly discouraged.

Directives that can be omitted from pass 2 are:

- `GBLA`, `GBLL`, `GBLS`.
- `LCLA`, `LCLL`, `LCLS`.
- `SETA`, `SETL`, `SETS`.
- `RN`, `RLIST`.
- `CN`, `CP`.
- `SN`, `DN`, `QN`.
- `EQU`.
- `MAP`, `FIELD`.
- `GET`, `INCLUDE`.
- `IF`, `ELSE`, `ELIF`, `ENDIF`.
- `WHILE`, `WEND`.
- `ASSERT`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- ATTR.
- COMMON.
- EXPORTAS.
- IMPORT.
- EXTERN.
- KEEP.
- MACRO, MEND, MEXIT.
- REQUIRE8.
- PRESERVE8.


Note

Macros that appear only in pass 1 and not in pass 2 must contain only these directives.

ASSERT directive appears in pass 1 only

The code in the following example assembles without error although the ASSERT directive does not appear in pass 2:

```
    AREA ||.text||,CODE

x   EQU 42
    IF :LNOT: :DEF: sym
        ASSERT x == 42
    ENDIF
sym EQU 1
    END
```

Use of ELSE and ELIF directives

Directives that appear in pass 2 but do not appear in pass 1 cause an assembly error. However, this does not cause an assembly error when using the ELSE and ELIF directives if their matching IF directive appears in pass 1. The following example assembles without error because the IF directive appears in pass 1:

```
    AREA ||.text||,CODE

x   EQU 42
    IF :DEF: sym
    ELSE
        ASSERT x == 42
    ENDIF
sym EQU 1
    END
```

Related information

Two pass assembler diagnostics on page 943
How the assembler works

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.5 ALIAS directive

The ALIAS directive creates an alias for a symbol.

Syntax

```
ALIAS <name>, <aliasname>
```

where:

<name>

is the name of the symbol to create an alias for.

<aliasname>

is the name of the alias to be created.

Usage

The symbol <name> must already be defined in the source file before creating an alias for it. Properties of <name> set by the EXPORT directive are not inherited by <aliasname>, so you must use EXPORT on <aliasname> if you want to make the alias available outside the current source file. Apart from the properties set by the EXPORT directive, <name> and <aliasname> are identical.

Correct example

```
baz

bar PROC
    BX lr
    ENDP
    ALIAS  bar,foo    ; foo is an alias for bar
    EXPORT bar
    EXPORT foo        ; foo and bar have identical properties
                      ; because foo was created using ALIAS
    EXPORT baz        ; baz and bar are not identical
                      ; because the size field of baz is not set
```

Incorrect example

```
    EXPORT bar

    IMPORT car
    ALIAS  bar,foo ; ERROR - bar is not defined yet
    ALIAS  car,boo ; ERROR - car is external
bar PROC
    BX lr
    ENDP
```

Related information

EXPORT or GLOBAL directive on page 993

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.6 ALIGN directive

The ALIGN directive aligns the current location to a specified boundary by padding with zeros or NOP instructions.

Syntax

```
ALIGN {<expr>{,<offset>{,<pad>{,<padsize>}}}}
```

where:

<expr>
is a numeric expression evaluating to any power of 2 from 2^0 to 2^{31}

<offset>
can be any numeric expression

<pad>
can be any numeric expression

<padsize>
can be 1, 2 or 4.

Operation

The current location is aligned to the next lowest address of the form:

```
<offset> + <n> * <expr>
```

<n> is any integer which the assembler selects to minimise padding.

If <expr> is not specified, ALIGN sets the current location to the next word (four byte) boundary. The unused space between the previous and the new current location are filled with:

- Copies of <pad>, if <pad> is specified.
- NOP instructions, if all the following conditions are satisfied:
  - <pad> is not specified.
  - The ALIGN directive follows A32 or T32 instructions.
  - The current section has the CODEALIGN attribute set on the AREA directive.
- Zeros otherwise.

<pad> is treated as a byte, halfword, or word, according to the value of <padsize>. If <padsize> is not specified, <pad> defaults to bytes in data sections, halfwords in T32 code, or words in A32 code.

Usage

Use ALIGN to ensure that your data and code is aligned to appropriate boundaries. This is typically required in the following circumstances:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The ADR T32 pseudo-instruction can only load addresses that are word aligned, but a label within T32 code might not be word aligned. Use ALIGN 4 to ensure four-byte alignment of an address within T32 code.
- Use ALIGN to take advantage of caches on some Arm® processors. For example, the Arm940T processor has a cache with 16-byte lines. Use ALIGN 16 to align function entries on 16-byte boundaries and maximize the efficiency of the cache.
- A label on a line by itself can be arbitrarily aligned. Following A32 code is word-aligned (T32 code is halfword aligned). The label therefore does not address the code correctly. Use ALIGN 4 (or ALIGN 2 for T32) before the label.

Alignment is relative to the start of the ELF section where the routine is located. The section must be aligned to the same, or coarser, boundaries. The ALIGN attribute on the AREA directive is specified differently.

Examples

```
        AREA    cacheable, CODE, ALIGN=3

rout1   ; code          ; aligned on 8-byte boundary
        ; code
        MOV     pc,lr   ; aligned only on 4-byte boundary
        ALIGN   8       ; now aligned on 8-byte boundary
rout2   ; code
```

In the following example, the ALIGN directive tells the assembler that the next instruction is word aligned and offset by 3 bytes. The 3 byte offset is counted from the previous word aligned address, resulting in the second DCB placed in the last byte of the same word and 2 bytes of padding are to be added.

```
        AREA    OffsetExample, CODE

        DCB     1       ; This example places the two bytes in the first
        ALIGN   4,3     ; and fourth bytes of the same word.
        DCB     1       ; The second DCB is offset by 3 bytes from the
                        ; first DCB.
```

In the following example, the ALIGN directive tells the assembler that the next instruction is word aligned and offset by 2 bytes. Here, the 2 byte offset is counted from the next word aligned address, so the value <n> is set to 1 (<n>=0 clashes with the third DCB). This time three bytes of padding are to be added.

```
       AREA    OffsetExample1, CODE

       DCB     1       ; In this example, n cannot be 0 because it
       DCB     1       ; clashes with the 3rd DCB. The assembler
       DCB     1       ; sets n to 1.
       ALIGN   4,2     ; The next instruction is word aligned and
       DCB     2       ; offset by 2.
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

In the following example, the DCB directive makes the PC misaligned. The ALIGN directive ensures that the label subroutine1 and the following instruction are word aligned.

```
        AREA    Example, CODE, READONLY

start   LDR     r6,=label1
        ; code
        MOV     pc,lr
label1  DCB     1       ; PC now misaligned
        ALIGN           ; ensures that subroutine1 addresses
subroutine1             ; the following instruction.
        MOV r5,#0x5
```

Related information

AREA directive on page 974

6.6.7 AREA directive

The AREA directive instructs the assembler to assemble a new code or data section.

Syntax

```
AREA <sectionname>{,<attr>}{,<attr>}...
```

Where:

<sectionname>

Is the name to give to the section. Sections are independent, named, indivisible chunks of code or data that the linker manipulates.

You can choose any name for your sections. However, names starting with a non-alphabetic character must be enclosed in bars or a missing section name error is generated. For example, |1_DataArea|.

Certain names are conventional. For example, |.text| is used for code sections that the C compiler produces, or for code sections that are otherwise associated with the C library.

<attr>

Are one or more comma-delimited section attributes. Valid attributes are:

ALIGN=<expression>

By default, ELF sections are aligned on a four-byte boundary. <expression> can have any integer value from 0-31. The section is aligned on a $2^{<expression>}$-byte boundary. For example, if expression is 10, the section is aligned on a 1KB boundary.

This attribute is not the same as the way that the ALIGN directive is specified.



Note

Do not use ALIGN=0 or ALIGN=1 for A32 code sections. Do not use ALIGN=0 for T32 code sections.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

ASSOC=<section>

`<section>` specifies an associated ELF section. `<sectionname>` must be included in any link that includes `<section>`

CODE

Contains machine instructions. `READONLY` is the default.

CODEALIGN

Causes `armasm` to insert `NOP` instructions when the `ALIGN` directive is used after A32 or T32 instructions within the section, unless the `ALIGN` directive specifies a different padding. `CODEALIGN` is the default for execute-only sections.

COMDEF


Note

This attribute is deprecated. Use the `COMGROUP` attribute.

Is a common section definition. This ELF section can contain code or data. It must be identical to any other section of the same name in other source files.

Identical ELF sections with the same name are overlaid in the same section of memory by the linker. If any are different, the linker generates a warning and does not overlay the sections.

COMGROUP=<symbol_name>

Is the signature that makes the `AREA` part of the named ELF section group. See the `GROUP=<symbol_name>` for more information. The `COMGROUP` attribute marks the ELF section group with the `GRP_COMDAT` flag.

COMMON

Is a common data section. You must not define any code or data in it. The section is initialized to zeros by the linker. All common sections with the same name are overlaid in the same section of memory by the linker. They do not all have to be the same size. The linker allocates as much space that the largest common section of each name requires.

DATA

Contains data, not instructions. `READWRITE` is the default.

EXECONLY

Indicates that the section is execute-only. Execute-only sections must also have the `CODE` attribute, and must not have any of the following attributes:

- `READONLY`.
- `READWRITE`.
- `DATA`.
- `ZEROALIGN`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

armasm faults if any of the following occur in an execute-only section:

- Explicit data definitions, for example DCD and DCB.
- Implicit data definitions, for example LDR r0, =0xaabbccdd.
- Literal pool directives, for example LTORG, if there is literal data to be emitted.
- INCBIN or SPACE directives.
- ALIGN directives, if padding with NOP instructions cannot achieve the required alignment. armasm implicitly applies the CODEALIGN attribute to sections with the EXECONLY attribute.

FINI_ARRAY

Sets the ELF type of the current area to SHT_FINI_ARRAY.

GROUP=<symbol_name>

Is the signature that makes the AREA part of the named ELF section group. The source file or a file that the source file includes must define the group. All AREAs with the same <symbol_name> signature are part of the same group. Sections within a group are kept or discarded together.

INIT_ARRAY

Sets the ELF type of the current area to SHT_INIT_ARRAY.

LINKORDER=<section>

Specifies a relative location for the current section in the image. This attribute ensures that the order of all the sections with the LINKORDER attribute, with respect to each other, is the same as the order of the corresponding named <sections> in the image.

MERGE=<n>

Indicates that the linker can merge the current section with other sections with the MERGE=<n> attribute. <n> is the size of the elements in the section, for example <n> is 1 for characters. You must not assume that the section is merged, because the attribute does not force the linker to merge the sections.

NOALLOC

Indicates that no memory on the target system is allocated to this area.

NOINIT

Indicates that the data section is uninitialized, or initialized to zero. It contains only space reservation directives SPACE or DCB, DCD, DCDU, DCQ, DCQU, DCW, or DCWU with initialized values of zero. You can decide at link time whether an area is uninitialized or zero-initialized.


Note

Arm® Compiler for Embedded does not support systems with ECC or parity protection where the memory is not initialized.

PREINIT_ARRAY

Sets the ELF type of the current area to SHT_PREINIT_ARRAY.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

READONLY

Indicates that this section must not be written to. This attribute is the default for Code areas.

READWRITE

Indicates that this section can be read from and written to. This attribute is the default for Data areas.

SECFLAGS=<n>

Adds one or more ELF flags, denoted by `<n>`, to the current section.

SECTYPE=<n>

Sets the ELF type of the current section to `<n>`.

STRINGS

Adds the `SHF_STRINGS` flag to the current section. To use the `STRINGS` attribute, you must also use the `MERGE=1` attribute. The contents of the section must be strings that are nul-terminated using the `DCB` directive.

ZEROALIGN

Causes `armasm` to insert zeros when the `ALIGN` directive is used after A32 or T32 instructions within the section, unless the `ALIGN` directive specifies a different padding. `ZEROALIGN` is the default for sections that are not execute-only.

Usage

Use the `AREA` directive to subdivide your source file into ELF sections. You can use the same name in more than one `AREA` directive. All areas with the same name are placed in the same ELF section. Only the attributes of the first `AREA` directive of a particular name are applied.

In general, Arm recommends that you use separate ELF sections for code and data. However, you can put data in code sections. Large programs can usually be conveniently divided into several code sections. Large independent data sets are also best placed in separate sections.

`AREA` directives define the scope of numeric local labels, optionally subdivided by `ROUT` directives.

There must be at least one `AREA` directive for an assembly.


Note

If the directive uses PC-relative expressions and is in any of the `PREINIT_ARRAY`, `FINI_ARRAY`, or `INIT_ARRAY` ELF sections, then `armasm` emits `R_ARM_TARGET1` relocations for the `DCD` and `DCDU` directives. You can override the relocation using the `RELOC` directive after each `DCD` or `DCDU` directive. If this relocation is used, read/write sections might become read-only sections at link time if permitted by the platform ABI.

Example

The following example defines a read-only code section named `Example`:

```
    AREA    Example,CODE,READONLY   ; An example code section.
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
            ; code
```

Related information

ALIGN directive on page 971
RELOC directive on page 1023
DCD and DCDU directives on page 984
ELF sections and the AREA directive on page 901
Image Structure and Generation on page 570

6.6.8 ARM or CODE32 directive

The `ARM` directive instructs the assembler to interpret subsequent instructions as A32 instructions, using either the UAL or the pre-UAL Arm assembler language syntax. `CODE32` is a synonym for `ARM`.



Note

Not supported for AArch64 state.

Syntax

```
ARM
```

Usage

In files that contain code using different instruction sets, the `ARM` directive must precede any A32 code.

If necessary, this directive also inserts up to three bytes of padding to align to the next word boundary.

This directive does not assemble to any instructions. It also does not change the state. It only instructs `armasm` to assemble A32 instructions as appropriate, and inserts padding if necessary.

Example

This example shows how you can use `ARM` and `THUMB` directives to switch state and assemble both A32 and T32 instructions in a single area.

```
        AREA ToT32, CODE, READONLY      ; Name this block of code

        ENTRY                           ; Mark first instruction to execute
        ARM                             ; Subsequent instructions are A32
start
        ADR     r0, into_t32 + 1        ; Processor starts in A32 state
        BX      r0                      ; Inline switch to T32 state
        THUMB                           ; Subsequent instructions are T32
into_t32
        MOVS    r0, #10                 ; New-style T32 instructions
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

CODE16 directive on page 981
THUMB directive on page 1030
Arm Architecture Reference Manuals

6.6.9 ASSERT directive

The ASSERT directive generates an error message during assembly if a given assertion is false.

Syntax

```
ASSERT <logical-expression>
```

Parameters

<logical-expression>

An assertion that can evaluate to either {TRUE} or {FALSE}.

Operation

Use ASSERT to ensure that any necessary condition is met during assembly.

If the assertion is false an error message is generated and assembly fails.

Example

```
ASSERT  label1 <= label2    ; Tests if the address

                            ; represented by label1
                            ; is <= the address
                            ; represented by label2.
```

Related information

INFO directive on page 1012

6.6.10 ATTR directive

The ATTR set directives set values for the ABI build attributes. The ATTR scope directives specify the scope for which the set value applies to.


Note

ATTR is supported only for AArch32 state.

Syntax

```
ATTR FILESCOPE
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
ATTR SCOPE <name>

ATTR <settype tagid>, <value>
```

Parameters

<name>

A section name or symbol name.

<settype>

Can be any of:

- SETVALUE.
- SETSTRING.
- SETCOMPATWITHVALUE.
- SETCOMPATWITHSTRING.

<tagid>

An attribute tag name (or its numerical value) defined in the ABI for the Arm® Architecture.

<value>

Depends on <settype>:

- A 32-bit integer value when <settype> is SETVALUE or SETCOMPATWITHVALUE.
- A null-terminated string when <settype> is SETSTRING or SETCOMPATWITHSTRING.

Operation

The ATTR set directives following the ATTR FILESCOPE directive apply to the entire object file. The ATTR set directives following the ATTR SCOPE <name> directive apply only to the named section or symbol.

For tags that expect an integer, you must use SETVALUE or SETCOMPATWITHVALUE. For tags that expect a string, you must use SETSTRING or SETCOMPATWITHSTRING.

Use SETCOMPATWITHVALUE and SETCOMPATWITHSTRING to set tag values which the object file is also compatible with.

Example: Set the value for Tag_VFP_arch

```
ATTR  SETSTRING Tag_CPU_raw_name, "Cortex-A8"
ATTR  SETVALUE  Tag_VFP_arch, 3   ; VFPv3 instructions permitted.
ATTR  SETVALUE  10, 3             ; 10 is the numerical value of
                                  ; Tag_VFP_arch.
```

Related information

Addenda to, and Errata in, the ABI for the Arm Architecture

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.11 CN directive

The CN directive defines a name for a coprocessor register.

Syntax

```
<name> CN <expr>
```

where:

<name>
is the name to be defined for the coprocessor register. <name> cannot be the same as any of the predefined names.

<expr>
evaluates to a coprocessor register number from 0 to 15.

Usage

Use CN to allocate convenient names to registers, to help you remember what you use each register for.


Note

Avoid conflicting uses of the same register under different names.

The names c0 to c15 are predefined.

Example

```
power    CN  6        ; defines power as a symbol for
                      ; coprocessor register 6
```

6.6.12 CODE16 directive

The CODE16 directive instructs the assembler to interpret subsequent instructions as T32 instructions, using the UAL syntax.


Note

Not supported for AArch64 state.

Syntax

```
CODE16
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Usage

In files that contain code using different instruction sets, `CODE16` must precede T32 code written in pre-UAL syntax.

If necessary, this directive also inserts one byte of padding to align to the next halfword boundary.

This directive does not assemble to any instructions. It also does not change the state. It only instructs `armasm` to assemble T32 instructions as appropriate, and inserts padding if necessary.

Related information

ARM or CODE32 directive on page 978
THUMB directive on page 1030

6.6.13 COMMON directive

The `COMMON` directive allocates a block of memory of the defined size, at the specified symbol.

Syntax

```
COMMON <symbol>{,<size>{,<alignment>}} {[<attr>]}
```

where:

<symbol>
is the symbol name. The symbol name is case-sensitive.

<size>
is the number of bytes to reserve.

<alignment>
is the alignment.

<attr>
can be any one of:

DYNAMIC
sets the ELF symbol visibility to `STV_DEFAULT`.

PROTECTED
sets the ELF symbol visibility to `STV_PROTECTED`.

HIDDEN
sets the ELF symbol visibility to `STV_HIDDEN`.

INTERNAL
sets the ELF symbol visibility to `STV_INTERNAL`.

Usage

You specify how the memory is aligned. If the alignment is omitted, the default alignment is four. If the size is omitted, the default size is zero.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

You can access this memory as you would any other memory, but no space is allocated by the assembler in object files. The linker allocates the required space as zero-initialized memory during the link stage.

You cannot define, IMPORT or EXTERN a symbol that has already been created by the COMMON directive. In the same way, if a symbol has already been defined or used with the IMPORT or EXTERN directive, you cannot use the same symbol for the COMMON directive.

Correct example

```
    LDR       r0, =xyz

    COMMON    xyz,255,4   ; defines 255 bytes of ZI store, word-aligned
```

Incorrect example

```
    COMMON    foo,4,4

    COMMON    bar,4,4
foo DCD       0         ; cannot define label with same name as COMMON
    IMPORT    bar       ; cannot import label with same name as COMMON
```

6.6.14 CP directive

The CP directive defines a name for a specified coprocessor.

Syntax

```
<name> CP <expr>
```

where:

<name>

is the name to be assigned to the coprocessor. <name> cannot be the same as any of the predefined names.

<expr>

evaluates to a coprocessor number within the range 0 to 15.

Usage

Use CP to allocate convenient names to coprocessors, to help you to remember what you use each one for.


Note

Avoid conflicting uses of the same coprocessor under different names.

The names p0 to p15 are predefined for coprocessors 0 to 15.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Example

```
dmu    CP  6       ; defines dmu as a symbol for
                   ; coprocessor 6
```

6.6.15 DATA directive

The DATA directive is no longer required. It is ignored by the assembler.

6.6.16 DCB directive

The DCB directive allocates one or more bytes of memory, and defines the initial runtime contents of the memory.

Syntax

label DCB <expr>{,<expr>}...

where:

<expr>

is either:

- A numeric expression that evaluates to an integer in the range -128 to 255.
- A quoted string. The characters of the string are loaded into consecutive bytes of store.

Usage

If DCB is followed by an instruction, use an ALIGN directive to ensure that the instruction is aligned.

= is a synonym for DCB.

Example

Unlike C strings, Arm® assembler strings are not nul-terminated. You can construct a nul-terminated C string using DCB as follows:

```
C_string   DCB  "C_string",0
```

Related information

Numeric expressions on page 956
DCD and DCDU directives on page 984
DCQ and DCQU directives on page 989
DCW and DCWU directives on page 990
SPACE and FILL directives on page 1029
ALIGN directive on page 971

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.17 DCD and DCDU directives

The DCD directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory. DCDU is the same, except that the memory alignment is arbitrary.

Syntax

```
label DCD{U} <expr>{,<expr>}
```

where:

<expr>

is either:

- A numeric expression.
- A PC-relative expression.

Usage

DCD inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment.

Use DCDU if you do not require alignment.

& is a synonym for DCD.

Examples

```
data1   DCD     1,5,20      ; Defines 3 words containing
                            ; decimal values 1, 5, and 20
data2   DCD     mem06 + 4   ; Defines 1 word containing 4 +
                            ; the address of the label mem06
        AREA    MyData, DATA, READWRITE
        DCB     255         ; Now misaligned ...
data3   DCDU    1,5,20      ; Defines 3 words containing
                            ; 1, 5 and 20, not word aligned
```

Related information

DCB directive on page 984
DCQ and DCQU directives on page 989
DCW and DCWU directives on page 990
SPACE and FILL directives on page 1029
Numeric expressions on page 956
DCI directive on page 987

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.18 DCDO directive

The DCDO directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory as an offset from the static base register, sb (R9).

Syntax

label DCDO <expr>{,<expr>}...

where:

<expr>
is a register-relative expression or label. The base register must be sb.

Usage

Use DCDO to allocate space in memory for static base register relative relocatable addresses.

Example

```
    IMPORT  externsym
    DCDO    externsym   ; 32-bit word relocated by offset of
                        ; externsym from base of SB section.
```

6.6.19 DCFD and DCFDU directives

The DCFD directive allocates memory for word-aligned double-precision floating-point numbers, and defines the initial runtime contents of the memory. DCFDU is the same, except that the memory alignment is arbitrary.

Syntax

label DCFD{U} <fpliteral>{,<fpliteral>}...

where:

<fpliteral>
is a double-precision floating-point literal.

Usage

Double-precision numbers occupy two words and must be word aligned to be used in arithmetic operations. The assembler inserts up to three bytes of padding before the first defined number, if necessary, to achieve four-byte alignment.

Use DCFDU if you do not require alignment.

The word order used when converting <fpliteral> to internal form is controlled by the floating-point architecture selected. You cannot use DCFD or DCFDU if you select the --fpu none option.

The range for double-precision numbers is:

- Maximum 1.79769313486231571e+308.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Minimum 2.22507385850720138e-308.

Examples

```
        DCFD    1E308,-4E-100
        DCFDU   10000,-.1,3.1E26
```

Related information

DCFS and DCFSU directives on page 987
Syntax of floating-point literals on page 958

6.6.20 DCFS and DCFSU directives

The DCFS directive allocates memory for word-aligned single-precision floating-point numbers, and defines the initial runtime contents of the memory. DCFSU is the same, except that the memory alignment is arbitrary.

Syntax

label DCFS{U} <fpliteral>{,<fpliteral>}...

where:

<fpliteral>
is a single-precision floating-point literal.

Usage

Single-precision numbers occupy one word and must be word aligned to be used in arithmetic operations. DCFS inserts up to three bytes of padding before the first defined number, if necessary to achieve four-byte alignment.

Use DCFSU if you do not require alignment.

The range for single-precision values is:

- Maximum 3.40282347e+38.
- Minimum 1.17549435e-38.

Examples

```
        DCFS    1E3,-4E-9
        DCFSU   1.0,-.1,3.1E6
```

Related information

DCFD and DCFDU directives on page 986
Syntax of floating-point literals on page 958

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.21 DCI directive

The DCI directive allocates memory that is aligned and defines the initial runtime contents of the memory.

In A32 code, it allocates one or more words of memory, aligned on four-byte boundaries.

In T32 code, it allocates one or more halfwords of memory, aligned on two-byte boundaries.

Syntax

```
label DCI{.W} <expr>{,<expr>}
```

where:

<expr>
is a numeric expression.

{.W}
if present, indicates that four bytes must be inserted in T32 code.

Usage

The DCI directive is very like the DCD or DCW directives, but the location is marked as code instead of data. Use DCI when writing macros for new instructions not supported by the version of the assembler you are using.

In A32 code, DCI inserts up to three bytes of padding before the first defined word, if necessary, to achieve four-byte alignment. In T32 code, DCI inserts an initial byte of padding, if necessary, to achieve two-byte alignment.

You can use DCI to insert a bit pattern into the instruction stream. For example, use:

```
    DCI 0x46c0
```

to insert the T32 operation MOV r8,r8.

Example macro

```
    MACRO           ; this macro translates newinstr Rd,Rm
                    ; to the appropriate machine code
    newinst     $Rd,$Rm
    DCI         0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm
    MEND
```

32-bit T32 example

```
    DCI.W   0xf3af8000   ; inserts 32-bit NOP, 2-byte aligned.
```

Related information

Numeric expressions on page 956
DCD and DCDU directives on page 984

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

DCW and DCWU directives on page 990

6.6.22 DCQ and DCQU directives

The DCQ directive allocates one or more eight-byte blocks of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory. DCQU is the same, except that the memory alignment is arbitrary.

Syntax

```
label DCQ{U} {-}<literal>{,{-}<literal>...}

label DCQ{U} <expr>{,<expr>...}
```

where:

<literal>

is a 64-bit numeric literal.

The range of numbers permitted is 0 to 2^{64}-1.

In addition to the characters normally permitted in a numeric literal, you can prefix <literal> with a minus sign. In this case, the range of numbers permitted is -2^{63} to -1.

The result of specifying -<n> is the same as the result of specifying 2^{64}-<n>.

<expr>

is either:

- A numeric expression.
- A PC-relative expression.


Note

armasm accepts expressions in DCQ and DCQU directives only when you are assembling for AArch64 targets.

Usage

DCQ inserts up to three bytes of padding before the first defined eight-byte block, if necessary, to achieve four-byte alignment.

Use DCQU if you do not require alignment.

Correct example

```
        AREA    MiscData, DATA, READWRITE
data    DCQ     -225,2_101      ; 2_101 means binary 101.
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Incorrect example

```
number  EQU     2               ; This code assembles for AArch64 targets only.

        DCQU    number          ; For AArch32 targets, DCQ and DCQU only accept
                                ; literals, not expressions.
```

Related information

DCB directive on page 984
DCD and DCDU directives on page 984
DCW and DCWU directives on page 990
SPACE and FILL directives on page 1029
Numeric expressions on page 956

6.6.23 DCW and DCWU directives

The DCW directive allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory. DCWU is the same, except that the memory alignment is arbitrary.

Syntax

```
label DCW{U} <expr>{,<expr>}...
```

where:

<expr>
is a numeric expression that evaluates to an integer in the range -32768 to 65535.

Usage

DCW inserts a byte of padding before the first defined halfword if necessary to achieve two-byte alignment.

Use DCWU if you do not require alignment.

Examples

```
data    DCW     -225,2*number   ; number must already be defined

        DCWU    number+4
```

Related information

DCB directive on page 984
DCD and DCDU directives on page 984
DCQ and DCQU directives on page 989
SPACE and FILL directives on page 1029
Numeric expressions on page 956

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.24 END directive

The END directive informs the assembler that it has reached the end of a source file.

Syntax

```
END
```

Usage

Every assembly language source file must end with END on a line by itself.

If the source file has been included in a parent file by a GET directive, the assembler returns to the parent file and continues assembly at the first line following the GET directive.

If END is reached in the top-level source file during the first pass without any errors, the second pass begins.

If END is reached in the top-level source file during the second pass, the assembler finishes the assembly and writes the appropriate output.

Related information

GET or INCLUDE directive on page 1007

6.6.25 ENDFUNC or ENDP directive

The ENDFUNC directive marks the end of an AAPCS-conforming function. ENDP is a synonym for ENDFUNC.

Related information

FUNCTION or PROC directive on page 1005

6.6.26 ENTRY directive

The ENTRY directive declares an entry point to a program.

Syntax

```
ENTRY
```



Note

The ENTRY directive is an armasm legacy assembler feature. If a GNU-syntax assembler file is to be the entry point for an image, then you must define the entry point at the link stage using the armlink option --entry.

Usage

A program must have an entry point. You can specify an entry point in the following ways:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- Using the ENTRY directive in armasm legacy assembly language source code.
- Providing a main() function in C or C++ source code.
- Using the armlink command-line option --entry.

You can declare more than one entry point in a program, although a source file cannot contain more than one ENTRY directive. For example, a program could contain multiple assembly language source files, each with an ENTRY directive. Or it could contain a C or C++ file with a main() function and one or more assembly source files with an ENTRY directive.

If the program contains multiple entry points, then you must select one of them. You do this by exporting the symbol for the ENTRY directive that you want to use as the entry point, then using the armlink option --entry to select the exported symbol.

Example

```
        AREA   ARMex, CODE, READONLY

        ENTRY      ; Entry point for the application.
        EXPORT ep1 ; Export the symbol so the linker can find it
ep1                ; in the object file.
        ; code
        END
```

When you invoke armlink, if other entry points are declared in the program, then you must specify --entry=ep1, to select ep1.

Related information

Image entry points on page 577
--entry=location on page 460

6.6.27 EQU directive

The EQU directive gives a symbolic name to a numeric constant, a register-relative value or a PC-relative value.

Syntax

```
<name> EQU <expr>{, <type>}
```

where:

<name>

is the symbolic name to assign to the value.

<expr>

is a register-relative address, a PC-relative address, an absolute address, or a 32-bit integer constant.

<type>

is optional. <type> can be any one of:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- ARM.
- THUMB.
- CODE32.
- CODE16.
- DATA.

You can use <type> only if <expr> is an absolute address. If <name> is exported, the <name> entry in the symbol table in the object file is marked as ARM, THUMB, CODE32, CODE16, or DATA, according to <type>. This can be used by the linker.

Usage

Use EQU to define constants. This is similar to the use of #define to define a constant in C.

* is a synonym for EQU.

Examples

```
abc EQU 2               ; Assigns the value 2 to the symbol abc.

xyz EQU label+8         ; Assigns the address (label+8) to the
                        ; symbol xyz.
fiq EQU 0x1C, CODE32    ; Assigns the absolute address 0x1C to
                        ; the symbol fiq, and marks it as code.
```

Related information

KEEP directive on page 1013
EXPORT or GLOBAL directive on page 993

6.6.28 EXPORT or GLOBAL directive

The EXPORT directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. GLOBAL is a synonym for EXPORT.

Syntax

```
EXPORT {[WEAK]}
EXPORT <symbol> {[SIZE=<n>]}
EXPORT <symbol> {[<type>{,<set>}]}
EXPORT <symbol> [<attr>{,<type>{,<set>}}{,SIZE=<n>}]
EXPORT <symbol> [WEAK {,<attr>}{,<type>{,<set>}}{,SIZE=<n>}]
```

Parameters

<symbol>

The symbol name to export. The symbol name is case-sensitive. If <symbol> is omitted, all symbols are exported.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

WEAK

<symbol> is only imported into other sources if no other source exports an alternative <symbol>. If [WEAK] is used without <symbol>, all exported symbols are weak.

<attr>

One of the following:

DYNAMIC

Sets the ELF symbol visibility to STV_DEFAULT.

PROTECTED

Sets the ELF symbol visibility to STV_PROTECTED.

HIDDEN

Sets the ELF symbol visibility to STV_HIDDEN.

INTERNAL

Sets the ELF symbol visibility to STV_INTERNAL.

<type>

The symbol type:

DATA

<symbol> is treated as data when the source is assembled and linked.

CODE

<symbol> is treated as code when the source is assembled and linked.

ELFTYPE=<n>

<symbol> is treated as a particular ELF symbol, as specified by the value of <n>, where <n> can be any number from 0 to 15.

If unspecified, the assembler determines the most appropriate type. Usually the assembler determines the correct type so you are not required to specify it.

<set>

Specifies the instruction set:

ARM

<symbol> is treated as an A32 symbol.

THUMB

<symbol> is treated as a T32 symbol.

If unspecified, the assembler determines the most appropriate set.

<n>

Specifies the size and can be any 32-bit value. If the SIZE attribute is not specified, the assembler calculates the size:

- For PROC and FUNCTION symbols, the size is set to the size of the code until its ENDP or ENDFUNC.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- For other symbols, the size is the size of instruction or data on the same source line. If there is no instruction or data, the size is zero.

Operation

Use `EXPORT` to give code in other files access to symbols in the current file.

Use the `[WEAK]` attribute to inform the linker that a different instance of `<symbol>` takes precedence over this one, if a different one is available from another source. You can use the `[WEAK]` attribute with any of the symbol visibility attributes.

Example: Export a function name

```
        AREA    Example,CODE,READONLY
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external modules.
DoAdd   ADD     r0,r0,r1
```

Example: Override symbol visibility for duplicate exports

In the following example, the last `EXPORT` takes precedence for both binding and visibility:

```
    EXPORT  SymA[WEAK]      ; Export as weak-hidden
    EXPORT  SymA[DYNAMIC]   ; SymA becomes non-weak dynamic.
```

Example: Use of the EXPORT SIZE attribute

The following examples show the use of the `SIZE` attribute:

```
    EXPORT symA [SIZE=4]
    EXPORT symA [DATA, SIZE=4]
```

Related information

IMPORT and EXTERN directives on page 1010
ELF for the Arm Architecture

6.6.29 EXPORTAS directive

The `EXPORTAS` directive enables you to export a symbol from the object file, corresponding to a different symbol in the source file.

Syntax

```
EXPORTAS <symbol1>, <symbol2>
```

where:

`<symbol1>`

is the symbol name in the source file. `<symbol1>` must have been defined already. It can be any symbol, including an area name, a label, or a constant.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<symbol2>
is the symbol name you want to appear in the object file.

The symbol names are case-sensitive.

Usage

Use EXPORTAS to change a symbol in the object file without having to change every instance in the source file.

Examples

```
    AREA data1, DATA       ; Starts a new area data1.
    AREA data2, DATA       ; Starts a new area data2.
    EXPORTAS data2, data1  ; The section symbol referred to as data2
                           ; appears in the object file string table as data1.
one EQU  2
    EXPORTAS one, two      ; The symbol 'two' appears in the object
    EXPORT one             ; file's symbol table with the value 2.
```

Related information

EXPORT or GLOBAL directive on page 993

6.6.30 FIELD directive

The FIELD directive describes space within a storage map that has been defined using the MAP directive.

Syntax

label FIELD <expr>

where:

<label>
is an optional label. If specified, <label> is assigned the value of the storage location counter, {VAR}. The storage location counter is then incremented by the value of <expr>.

<expr>
is an expression that evaluates to the number of bytes to increment the storage counter.

Usage

If a storage map is set by a MAP directive that specifies a <base-register>, the base register is implicit in all labels defined by following FIELD directives, until the next MAP directive. These register-relative labels can be quoted in load and store instructions.

is a synonym for FIELD.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Examples

The following example shows how register-relative labels are defined using the `MAP` and `FIELD` directives:

```
    MAP     0,r9        ; set {VAR} to the address stored in R9

    FIELD   4           ; increment {VAR} by 4 bytes
Lab FIELD   4           ; set Lab to the address [R9 + 4]
                        ; and then increment {VAR} by 4 bytes
    LDR     r0,Lab      ; equivalent to LDR r0,[r9,#4]
```

When using the `MAP` and `FIELD` directives, you must ensure that the values are consistent in both passes. The following example shows a use of `MAP` and `FIELD` that causes inconsistent values for the symbol `x`. In the first pass `sym` is not defined, so `x` is at `0x04` + `R9`. In the second pass, `sym` is defined, so `x` is at `0x00` + `R0`. This example results in an assembly error.

```
    MAP 0, r0

    if :LNOT: :DEF: sym
      MAP 0, r9
      FIELD 4  ;  x   is at 0x04+R9   in first pass
    ENDIF
x   FIELD 4    ; x   is at 0x00+R0   in second pass
sym LDR r0, x  ; inconsistent values for x results in assembly error
```

Related information

MAP directive on page 1019
Directives that can be omitted in pass 2 of the assembler on page 969
How the assembler works

6.6.31 FRAME ADDRESS directive

The `FRAME ADDRESS` directive describes how to calculate the canonical frame address for the following instructions.

Syntax

```
FRAME ADDRESS <reg>{,<offset>}
```

where:

`<reg>`

is the register on which the canonical frame address is to be based. This is SP unless the function uses a separate frame pointer.

`<offset>`

is the offset of the canonical frame address from `<reg>`. If `<offset>` is zero, you can omit it.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Usage

Use `FRAME ADDRESS` if your code alters which register the canonical frame address is based on, or if it changes the offset of the canonical frame address from the register. You must use `FRAME ADDRESS` immediately after the instruction that changes the calculation of the canonical frame address.

You can only use `FRAME ADDRESS` in functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.


Note

If your code uses a single instruction to save registers and alter the stack pointer, you can use `FRAME PUSH` instead of using both `FRAME ADDRESS` and `FRAME SAVE`.

If your code uses a single instruction to load registers and alter the stack pointer, you can use `FRAME POP` instead of using both `FRAME ADDRESS` and `FRAME RESTORE`.

Example

```
_fn     FUNCTION        ; CFA (Canonical Frame Address) is value

                        ; of SP on entry to function
        PUSH    {r4,fp,ip,lr,pc}
        FRAME PUSH {r4,fp,ip,lr,pc}
        SUB     sp,sp,#4            ; CFA offset now changed
        FRAME ADDRESS sp,24         ; - so we correct it
        ADD     fp,sp,#20
        FRAME ADDRESS fp,4          ; New base register
        ; code using fp to base call-frame on, instead of SP
```

Related information

FRAME POP directive on page 998
FRAME PUSH directive on page 999

6.6.32 FRAME POP directive

The `FRAME POP` directive informs the assembler when the callee reloads registers.

Syntax

There are the following alternative syntaxes for `FRAME POP`:

```
FRAME POP {\<reglist>\}

FRAME POP {\<reglist>\},<n>

FRAME POP <n>
```

where:

`<reglist>`
is a list of registers restored to the values they had on entry to the function. There must be at least one register in the list.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<n>
is the number of bytes that the stack pointer moves.

Usage

`FRAME POP` is equivalent to a `FRAME ADDRESS` and a `FRAME RESTORE` directive. You can use it when a single instruction loads registers and alters the stack pointer.

You must use `FRAME POP` immediately after the instruction it refers to.

You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives. You do not have to do this after the last instruction in a function.

If `<n>` is not specified or is zero, the assembler calculates the new offset for the canonical frame address from *`reglist`*. It assumes that:

- Each AArch32 register popped occupies four bytes on the stack.
- Each VFP single-precision register popped occupies four bytes on the stack, plus an extra four-byte word for each list.
- Each VFP double-precision register popped occupies eight bytes on the stack, plus an extra four-byte word for each list.

Related information

FRAME ADDRESS directive on page 997
FRAME PUSH directive on page 999
FRAME RESTORE directive on page 1001

6.6.33 FRAME PUSH directive

The `FRAME PUSH` directive informs the assembler when the callee saves registers, normally at function entry.

Syntax

There are the following alternative syntaxes for `FRAME PUSH`:

```
FRAME PUSH {\<reglist>\}

FRAME PUSH {\<reglist>\},<n>

FRAME PUSH <n>
```

where:

<reglist>
is a list of registers stored consecutively below the canonical frame address. There must be at least one register in the list.

<n>
is the number of bytes that the stack pointer moves.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Usage

`FRAME PUSH` is equivalent to a `FRAME ADDRESS` and a `FRAME SAVE` directive. You can use it when a single instruction saves registers and alters the stack pointer.

You must use `FRAME PUSH` immediately after the instruction it refers to.

You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

If `<n>` is not specified or is zero, the assembler calculates the new offset for the canonical frame address from `<reglist>`. It assumes that:

- Each AArch32 register pushed occupies four bytes on the stack.
- Each VFP single-precision register pushed occupies four bytes on the stack, plus an extra four-byte word for each list.
- Each VFP double-precision register popped occupies eight bytes on the stack, plus an extra four-byte word for each list.

Example

```
p   PROC ; Canonical frame address is SP + 0

    EXPORT  p
    PUSH    {r4-r6,lr}
         ; SP has moved relative to the canonical frame address,
         ; and registers R4, R5, R6 and LR are now on the stack
    FRAME PUSH {r4-r6,lr}
         ; Equivalent to:
         ; FRAME ADDRESS    sp,16       ; 16 bytes in {R4-R6,LR}
         ; FRAME SAVE    {r4-r6,lr},-16
```

Related information

FRAME ADDRESS directive on page 997
FRAME POP directive on page 998
FRAME SAVE directive on page 1002

6.6.34 FRAME REGISTER directive

The `FRAME REGISTER` directive maintains a record of the locations of function arguments held in registers.

Syntax

```
FRAME REGISTER <reg1>, <reg2>
```

where:

`<reg1>`
is the register that held the argument on entry to the function.

`<reg2>`
is the register in which the value is preserved.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Usage

Use the `FRAME REGISTER` directive when you use a register to preserve an argument that was held in a different register on entry to a function.

You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

6.6.35 FRAME RESTORE directive

The `FRAME RESTORE` directive informs the assembler that the contents of specified registers have been restored to the values they had on entry to the function.

Syntax

```
FRAME RESTORE {\<reglist>\}
```

where:

`<reglist>`

is a list of registers whose contents have been restored. There must be at least one register in the list.

Usage

You can only use `FRAME RESTORE` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives. Use it immediately after the callee reloads registers from the stack. You do not have to do this after the last instruction in a function.

`<reglist>` can contain integer registers or floating-point registers, but not both.



Note

If your code uses a single instruction to load registers and alter the stack pointer, you can use `FRAME POP` instead of using both `FRAME RESTORE` and `FRAME ADDRESS`.

Related information

FRAME POP directive on page 998

6.6.36 FRAME RETURN ADDRESS directive

The `FRAME RETURN ADDRESS` directive provides for functions that use a register other than LR for their return address.

Syntax

```
FRAME RETURN ADDRESS <reg>
```

where:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<reg>
is the register used for the return address.

Usage

Use the `FRAME RETURN ADDRESS` directive in any function that does not use LR for its return address. Otherwise, a debugger cannot backtrace through the function.

You can only use `FRAME RETURN ADDRESS` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives. Use it immediately after the `FUNCTION` or `PROC` directive that introduces the function.


Note

Any function that uses a register other than LR for its return address is not AAPCS compliant. Such a function must not be exported.

6.6.37 FRAME SAVE directive

The `FRAME SAVE` directive describes the location of saved register contents relative to the canonical frame address.

Syntax

```
FRAME SAVE {\<reglist>\}, <offset>
```

where:

<reglist>
is a list of registers stored consecutively starting at `<offset>` from the canonical frame address. There must be at least one register in the list.

Usage

You can only use `FRAME SAVE` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Use it immediately after the callee stores registers onto the stack.

`<reglist>` can include registers which are not required for backtracing. The assembler determines which registers it requires to record in the DWARF call frame information.


Note

If your code uses a single instruction to save registers and alter the stack pointer, you can use `FRAME PUSH` instead of using both `FRAME SAVE` and `FRAME ADDRESS`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

FRAME PUSH directive on page 999

6.6.38 FRAME STATE REMEMBER directive

The `FRAME STATE REMEMBER` directive saves the current information on how to calculate the canonical frame address and locations of saved register values.

Syntax

```
FRAME STATE REMEMBER
```

Usage

During an inline exit sequence the information about calculation of canonical frame address and locations of saved register values can change. After the exit sequence another branch can continue using the same information as before. Use `FRAME STATE REMEMBER` to preserve this information, and `FRAME STATE RESTORE` to restore it.

These directives can be nested. Each `FRAME STATE RESTORE` directive must have a corresponding `FRAME STATE REMEMBER` directive.

You can only use `FRAME STATE REMEMBER` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Example

```
        ; function code

        FRAME STATE REMEMBER
            ; save frame state before in-line exit sequence
        POP     {r4-r6,pc}
            ; do not have to FRAME POP here, as control has
            ; transferred out of the function
        FRAME STATE RESTORE
            ; end of exit sequence, so restore state
exitB   ; code for exitB
        POP     {r4-r6,pc}
        ENDP
```

Related information

FRAME STATE RESTORE directive on page 1003
FUNCTION or PROC directive on page 1005

6.6.39 FRAME STATE RESTORE directive

The `FRAME STATE RESTORE` directive restores information about how to calculate the canonical frame address and locations of saved register values.

Syntax

```
FRAME STATE RESTORE
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Usage

You can only use `FRAME STATE RESTORE` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Related information

FRAME STATE REMEMBER directive on page 1003
FUNCTION or PROC directive on page 1005

6.6.40 FRAME UNWIND ON directive

The `FRAME UNWIND ON` directive instructs the assembler to produce unwind tables for this and subsequent functions.

Syntax

```
FRAME UNWIND ON
```

Usage

You can use this directive outside functions. In this case, the assembler produces unwind tables for all following functions until it reaches a `FRAME UNWIND OFF` directive.



Note

A `FRAME UNWIND` directive is not sufficient to turn on exception table generation. Furthermore a `FRAME UNWIND` directive, without other `FRAME` directives, is not sufficient information for the assembler to generate the unwind information.

Related information

--exceptions, --no_exceptions (armasm) on page 877
--exceptions_unwind, --no_exceptions_unwind on page 878

6.6.41 FRAME UNWIND OFF directive

The `FRAME UNWIND OFF` directive instructs the assembler to produce no unwind tables for this and subsequent functions.

Syntax

```
FRAME UNWIND OFF
```

Usage

You can use this directive outside functions. In this case, the assembler produces no unwind tables for all following functions until it reaches a `FRAME UNWIND ON` directive.

Related information

--exceptions, --no_exceptions (armasm) on page 877
--exceptions_unwind, --no_exceptions_unwind on page 878

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.42 FUNCTION or PROC directive

The FUNCTION directive marks the start of a function. PROC is a synonym for FUNCTION.

Syntax

```
<label> FUNCTION [{\<reglist1>\} [, {\<reglist2>\}]]
```

where:

<reglist1>
is an optional list of callee-saved AArch32 registers. If <reglist1> is not present, and your debugger checks register usage, it assumes that the AAPCS is in use. If you use empty brackets, this informs the debugger that all AArch32 registers are caller-saved.

<reglist2>
is an optional list of callee-saved VFP registers. If you use empty brackets, this informs the debugger that all VFP registers are caller-saved.

Usage

Use FUNCTION to mark the start of functions. The assembler uses FUNCTION to identify the start of a function when producing DWARF call frame information for ELF.

FUNCTION sets the canonical frame address to be R13 (SP), and the frame state stack to be empty.

Each FUNCTION directive must have a matching ENDFUNC directive. You must not nest FUNCTION and ENDFUNC pairs, and they must not contain PROC or ENDP directives.

You can use the optional <reglist> parameters to inform the debugger about an alternative procedure call standard, if you are using your own. Not all debuggers support this feature. See your debugger documentation for details.

If you specify an empty <reglist>, using {} , this indicates that all registers for the function are caller-saved. Typically you do this when writing a reset vector where the values in all registers are unknown on execution. This avoids problems in a debugger if it tries to construct a backtrace from the values in the registers.


Note

FUNCTION does not automatically cause alignment to a word boundary (or halfword boundary for T32). Use ALIGN if necessary to ensure alignment, otherwise the call frame might not point to the start of the function.

Examples

```
        ALIGN        ; Ensures alignment.

dadd    FUNCTION     ; Without the ALIGN directive this might not be word-aligned.
        EXPORT  dadd
        PUSH       {r4-r6,lr}    ; This line automatically word-aligned.
        FRAME PUSH {r4-r6,lr}
        ; subroutine body
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
        POP         {r4-r6,pc}
        ENDFUNC
func6   PROC {r4-r8,r12},{D1-D3} ; Non-AAPCS-conforming function.
        ...
        ENDP
func7   FUNCTION {}  ; Another non-AAPCS-conforming function.
        ...
        ENDFUNC
```

Related information

FRAME STATE RESTORE directive on page 1003
FRAME ADDRESS directive on page 997
ALIGN directive on page 971

6.6.43 GBLA, GBLL, and GBLS directives

The GBLA, GBLL, and GBLS directives declare and initialize global variables.

Syntax

```
<gblx> <variable>
```

where:

<gblx>

is one of GBLA, GBLL, or GBLS.

<variable>

is the name of the variable. <variable> must be unique among symbols within a source file.

Usage

The GBLA directive declares a global arithmetic variable, and initializes its value to 0.

The GBLL directive declares a global logical variable, and initializes its value to {FALSE}.

The GBLS directive declares a global string variable and initializes its value to a null string, "".

Using one of these directives for a variable that is already defined re-initializes the variable.

The scope of the variable is limited to the source file that contains it.

Set the value of the variable with a SETA, SETL, or SETS directive.

Global variables can also be set with the --predefine assembler command-line option.

Examples

The following example declares a variable objectsize, sets the value of objectsize to FF, and then uses it later in a SPACE directive:

```
            GBLA    objectsize    ; declare the variable name
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
objectsize  SETA    0xFF          ; set its value
            .
            .                     ; other code
            .
            SPACE   objectsize    ; quote the variable
```

The following example shows how to declare and set a variable when you invoke `armasm`. Use this when you want to set the value of a variable at assembly time. `--pd` is a synonym for `--predefine`.

```
armasm --cpu=8-A.32 --predefine "objectsize SETA 0xFF" sourcefile -o objectfile
```

Related information

LCLA, LCLL, and LCLS directives on page 1014
SETA, SETL, and SETS directives on page 1028
--predefine "directive" on page 891

6.6.44 GET or INCLUDE directive

The `GET` directive includes a file within the file being assembled. The included file is assembled at the location of the `GET` directive. `INCLUDE` is a synonym for `GET`.

Syntax

```
GET <filename>
```

where:

`<filename>`
is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or Windows format.

Operation

`GET` is useful for including macro definitions, `EQU`s, and storage maps in an assembly. When assembly of the included file is complete, assembly continues at the line following the `GET` directive.

By default the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the `-i` assembler command-line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes (" ").

The included file can contain additional `GET` directives to include other files.

If the included file is in a different directory from the current place, this becomes the current place until the end of the included file. The previous current place is then restored.

You cannot use `GET` to include object files.

Examples

```
    AREA    Example, CODE, READONLY
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    GET    file1.s                  ; includes file1 if it exists in the current
 place
    GET    c:\project\file2.s       ; includes file2
    GET    c:\Program files\file3.s ; space is permitted
```

Related information

INCBIN directive on page 1012
About armasm assembly language control directives on page 967

6.6.45 IF, ELSE, ENDIF, and ELIF directives

The `IF`, `ELSE`, `ENDIF`, and `ELIF` directives allow you to conditionally assemble sequences of instructions and directives.

Syntax

```
    IF <logical-expression>
        ... ;code
    {ELSE
        ... ;code}
    ENDIF
```

where:

`<logical-expression>`

is an expression that evaluates to either `{TRUE}` or `{FALSE}`.

Usage

Use `IF` with `ENDIF`, and optionally with `ELSE`, for sequences of instructions or directives that are only to be assembled or acted on under a specified condition.

`IF...ENDIF` conditions can be nested.

The `IF` directive introduces a condition that controls whether to assemble a sequence of instructions and directives. `[` is a synonym for `IF`.

The `ELSE` directive marks the beginning of a sequence of instructions or directives that you want to be assembled if the preceding condition fails. `|` is a synonym for `ELSE`.

The `ENDIF` directive marks the end of a sequence of instructions or directives that you want to be conditionally assembled. `]` is a synonym for `ENDIF`.

The `ELIF` directive creates a structure equivalent to `ELSE IF`, without the requirement for nesting or repeating the condition.

Using ELIF

Without using `ELIF`, you can construct a nested set of conditional instructions like this:

```
    IF <logical-expression>
        instructions
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    ELSE
        IF <logical-expression2>
            instructions
        ELSE
            IF <logical-expression3>
                instructions
            ENDIF
        ENDIF
    ENDIF
```

A nested structure like this can be nested up to 256 levels deep.

You can write the same structure more simply using `ELIF`:

```
    IF <logical-expression>
        instructions
    ELIF <logical-expression2>
        instructions
    ELIF <logical-expression3>
        instructions
    ENDIF
```

This structure only adds one to the current nesting depth, for the `IF...ENDIF` pair.

Examples

The following example assembles the first set of instructions if `NEWVERSION` is defined, or the alternative set otherwise:

Assembly conditional on a variable being defined

```
    IF :DEF:NEWVERSION
        ; first set of instructions or directives
    ELSE
        ; alternative set of instructions or directives
    ENDIF
```

Invoking `armasm` as follows defines `NEWVERSION`, so the first set of instructions and directives are assembled:

```
armasm --cpu=8-A.32 --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking `armasm` as follows leaves `NEWVERSION` undefined, so the second set of instructions and directives are assembled:

```
armasm --cpu=8-A.32 test.s
```

The following example assembles the first set of instructions if `NEWVERSION` has the value `{TRUE}`, or the alternative set otherwise:

Assembly conditional on a variable value

```
    IF NEWVERSION = {TRUE}
        ; first set of instructions or directives
    ELSE
        ; alternative set of instructions or directives
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    ENDIF
```

Invoking `armasm` as follows causes the first set of instructions and directives to be assembled:

```
armasm --cpu=8-A.32 --predefine "NEWVERSION SETL {TRUE}" test.s
```

Invoking `armasm` as follows causes the second set of instructions and directives to be assembled:

```
armasm --cpu=8-A.32 --predefine "NEWVERSION SETL {FALSE}" test.s
```

Related information

Relational operators on page 963
About armasm assembly language control directives on page 967

6.6.46 IMPORT and EXTERN directives

The `IMPORT` and `EXTERN` directives provide the assembler with a name that is not defined in the current assembly.

Syntax

```
<directive> <symbol> {[SIZE=<n>]}
<directive> <symbol> {[<type>]}
<directive> <symbol> [<attr>{,<type>}{,SIZE=<n>}]
<directive> <symbol> [WEAK <,attr>{,<type>}{,SIZE=<n>}]
```

Parameters

<directive>

Can be either:

IMPORT

Imports the symbol unconditionally.

EXTERN

Imports the symbol only if it is referred to in the current assembly.

<symbol>

A symbol name defined in a separately assembled source file, object file, or library. The symbol name is case-sensitive.

WEAK

Prevents the linker generating an error message if the symbol is not defined elsewhere. It also prevents the linker searching libraries that are not already included.

<attr>

Can be any one of:

DYNAMIC

Sets the ELF symbol visibility to `STV_DEFAULT`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

PROTECTED

Sets the ELF symbol visibility to `STV_PROTECTED`.

HIDDEN

Sets the ELF symbol visibility to `STV_HIDDEN`.

INTERNAL

Sets the ELF symbol visibility to `STV_INTERNAL`.

<type>

Specifies the symbol type:

DATA

`<symbol>` is treated as data when the source is assembled and linked.

CODE

`<symbol>` is treated as code when the source is assembled and linked.

ELFTYPE=<n>

`<symbol>` is treated as a particular ELF symbol, as specified by the value of `<n>`, where `<n>` can be any number from `0` to `15`.

If unspecified, the linker determines the most appropriate type.

<n>

Specifies the size and can be any 32-bit value. If the `SIZE` attribute is not specified, the assembler calculates the size:

- For `PROC` and `FUNCTION` symbols, the size is set to the size of the code until its `ENDP` or `ENDFUNC`.
- For other symbols, the size is the size of instruction or data on the same source line. If there is no instruction or data, the size is zero.

Operation

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If `[WEAK]` is not specified, the linker generates an error if no corresponding symbol is found at link time.

If `[WEAK]` is specified and no corresponding symbol is found at link time:

- If the reference is the destination of a `B` or `BL` instruction, the value of the symbol is taken as the address of the following instruction. This makes the `B` or `BL` instruction effectively a `NOP`.
- Otherwise, the value of the symbol is taken as zero.

Example: Test to see if the C++ library has been linked

The example tests to see if the C++ library has been linked, and branches conditionally on the result.

```
    AREA    Example, CODE, READONLY
    EXTERN  __CPP_INITIALIZE[WEAK]  ; If C++ library linked, gets the
                                    ; address of __CPP_INITIALIZE
                                    ; function.
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    LDR     r0,=__CPP_INITIALIZE    ; If not linked, address is zeroed.
    CMP     r0,#0                   ; Test if zero.
    BEQ     nocplusplus             ; Branch on the result.
```

Example: Use of the EXTERN SIZE attribute

The following examples show the use of the SIZE attribute:

```
    EXTERN symA [SIZE=4]
    EXTERN symA [DATA, SIZE=4]
```

Related information

EXPORT or GLOBAL directive on page 993
ELF for the Arm Architecture

6.6.47 INCBIN directive

The INCBIN directive includes a file within the file being assembled. The file is included as it is, without being assembled.

Syntax

```
INCBIN <filename>
```

where:

<filename>

is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or Windows format.

Operation

You can use INCBIN to include data, such as executable files, literals, or any arbitrary data. The contents of the file are added to the current ELF section, byte for byte, without being interpreted in any way. Assembly continues at the line following the INCBIN directive.

By default, the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the -i assembler command-line option to add directories to the search path. File names and directory names containing spaces must not be enclosed in double quotes (" ").

Example

```
    AREA    Example, CODE, READONLY
    INCBIN  file1.dat             ; Includes file1 if it exists in the current
 place
    INCBIN  c:\project\file2.txt   ; Includes file2.
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.48 INFO directive

The INFO directive supports diagnostic generation on either pass of the assembly.

Syntax

```
INFO {<numeric-expression>, <string-expression>{, <severity>}
```

where:

<numeric-expression>

is a numeric expression that is evaluated during assembly. If the expression evaluates to zero:

- No action is taken during pass one.
- <string-expression> is printed as a warning during pass two if <severity> is 1.
- <string-expression> is printed as a message during pass two if <severity> is 0 or not specified.

If the expression does not evaluate to zero:

- <string-expression> is printed as an error message and the assembly fails irrespective of whether <severity> is specified or not (non-zero values for <severity> are reserved in this case).

<string-expression>

is an expression that evaluates to a string.

<severity>

is an optional number that controls the severity of the message. Its value can be either 0 or 1. All other values are reserved.

Usage

INFO provides a flexible means of creating custom error messages.

! is very similar to INFO, but has less detailed reporting.

Examples

```
    INFO    0, "Version 1.0"

    IF endofdata <= label1
        INFO    4, "Data overrun at label1"
    ENDIF
```

Related information

ASSERT directive on page 979
String expressions on page 955
Numeric expressions on page 956

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.49 KEEP directive

The KEEP directive instructs the assembler to retain named local labels in the symbol table in the object file.

Syntax

```
KEEP label
```

where:

<label>

is the name of the local label to keep. If <label> is not specified, all named local labels are kept except register-relative labels.

Usage

By default, the only labels that the assembler describes in its output object file are:

- Exported labels.
- Labels that are relocated against.

Use KEEP to preserve local labels. This can help when debugging. Kept labels appear in the Arm® debuggers and in linker map files.

KEEP cannot preserve register-relative labels or numeric local labels.

Example

```
label   ADC     r2,r3,r4

        KEEP    label       ; makes label available to debuggers
        ADD     r2,r2,r5
```

Related information

MAP directive on page 1019
Numeric local labels on page 953

6.6.50 LCLA, LCLL, and LCLS directives

The LCLA, LCLL, and LCLS directives declare and initialize local variables.

Syntax

```
<lclx> <variable>
```

where:

<lclx>

is one of LCLA, LCLL, or LCLS.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<variable>
is the name of the variable. <variable> must be unique within the macro that contains it.

Usage

The LCLA directive declares a local arithmetic variable, and initializes its value to 0.

The LCLL directive declares a local logical variable, and initializes its value to {FALSE}.

The LCLS directive declares a local string variable, and initializes its value to a null string, "".

Using one of these directives for a variable that is already defined re-initializes the variable.

The scope of the variable is limited to a particular instantiation of the macro that contains it.

Set the value of the variable with a SETA , SETL, or SETS directive.

Example

```
        MACRO                           ; Declare a macro

$label  message $a                      ; Macro prototype line
        LCLS    err                     ; Declare local string
                                        ; variable err.
err     SETS    "error no: "            ; Set value of err
$label  ; code
        INFO    0, "err":CC::STR:$a     ; Use string
        MEND
```

Related information

GBLA, GBLL, and GBLS directives on page 1006
SETA, SETL, and SETS directives on page 1028

6.6.51 LTORG directive

The LTORG directive instructs the assembler to assemble the current literal pool immediately.

Syntax

```
LTORG
```

Usage

The assembler assembles the current literal pool at the end of every code section. The end of a code section is determined by the AREA directive at the beginning of the following section, or the end of the assembly.

These default literal pools can sometimes be out of range of some LDR , VLDR, and WLDR pseudo-instructions. Use LTORG to ensure that a literal pool is assembled within range.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Large programs can require several literal pools. Place LTORG directives after unconditional branches or subroutine return instructions so that the processor does not attempt to execute the constants as instructions.

The assembler word-aligns data in literal pools.

Example

```
        AREA    Example, CODE, READONLY
start   BL      func1
func1                           ; function body
        ; code
        LDR     r1,=0x55555555  ; => LDR R1, [pc, #offset to Literal Pool 1]
        ; code
        MOV     pc,lr           ; end function
        LTORG                   ; Literal Pool 1 contains literal &55555555.
data    SPACE   4200            ; Clears 4200 bytes of memory starting at current
 location.
        END                     ; Default literal pool is empty.
```

6.6.52 MACRO and MEND directives

The MACRO directive marks the start of the definition of a macro. Macro expansion terminates at the MEND directive.

Syntax

These two directives define a macro. The syntax is:

```
     MACRO

{$label}  macroname{$cond} {$parameter{,$parameter}...}
     ; code
     MEND
```

where:

$<label>

is a parameter that is substituted with a symbol given when the macro is invoked. The symbol is usually a label.

<macroname>

is the name of the macro. It must not begin with an instruction or directive name.

$<cond>

is a special parameter designed to contain a condition code. Values other than valid condition codes are permitted.

$<parameter>

is a parameter that is substituted when the macro is invoked. A default value for a parameter can be set using this format:

```
$<parameter>="<default value>"
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Double quotes must be used if there are any spaces within, or at either end of, the default value.

Usage

If you start any `WHILE...WEND` loops or `IF...ENDIF` conditions within a macro, they must be closed before the `MEND` directive is reached. You can use `MEXIT` to enable an early exit from a macro, for example, from within a loop.

Within the macro body, parameters such as `$<label>`, `$<parameter>` or `$<cond>` can be used in the same way as other variables. They are given new values each time the macro is invoked. Parameters must begin with `$` to distinguish them from ordinary symbols. Any number of parameters can be used.

`$<label>` is optional. It is useful if the macro defines internal labels. It is treated as a parameter to the macro. It does not necessarily represent the first instruction in the macro expansion. The macro defines the locations of any labels.

Use `|` as the argument to use the default value of a parameter. An empty string is used if the argument is omitted.

In a macro that uses several internal labels, it is useful to define each internal label as the base label with a different suffix.

Use a dot between a parameter and following text, or a following parameter, if a space is not required in the expansion. Do not use a dot between preceding text and a parameter.

You can use the `$<cond>` parameter for condition codes. Use the unary operator `:REVERSE_CC:` to find the inverse condition code, and `:CC_ENCODING:` to find the 4-bit encoding of the condition code.

Macros define the scope of local variables.

Macros can be nested.

Examples

A macro that uses internal labels to implement loops:

```
 ; macro definition

                MACRO                   ; start macro definition
$label          xmac    $p1,$p2
                ; code
$label.loop1    ; code
                ; code
                BGE     $label.loop1
$label.loop2    ; code
                BL      $p1
                BGT     $label.loop2
                ; code
                ADR     $p2
                ; code
                MEND                    ; end macro definition
 ; macro invocation
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
abc             xmac    subr1,de      ; invoke macro
                ; code                ; this is what is
abcloop1        ; code                ; is produced when
                ; code                ; the xmac macro is
                BGE     abcloop1      ; expanded
abcloop2        ; code
                BL      subr1
                BGT     abcloop2
                ; code
                ADR     de
                ; code
```

A macro that produces assembly-time diagnostics:

```
       MACRO                        ; Macro definition

       diagnose  $param1="default"  ; This macro produces
       INFO      0,"$param1"        ; assembly-time diagnostics
       MEND                         ; (on second assembly pass)
; macro expansion
       diagnose            ; Prints blank line at assembly-time
       diagnose "hello"    ; Prints "hello" at assembly-time
       diagnose |          ; Prints "default" at assembly-time
```

When variables are being passed in as arguments, use of | might leave some variables unsubstituted. To work around this, define the | in a LCLS or GBLS variable and pass this variable as an argument instead of |. For example:

```
        MACRO                ; Macro definition

        m2 $a,$b=r1,$c       ; The default value for $b is r1
        add $a,$b,$c         ; The macro adds $b and $c and puts result in $a.
        MEND                 ; Macro end
        MACRO                ; Macro definition
        m1 $a,$b             ; This macro adds $b to r1 and puts result in $a.
        LCLS def             ; Declare a local string variable for |
def     SETS "|"             ; Define |
        m2 $a,$def,$b        ; Invoke macro m2 with $def instead of |
                             ; to use the default value for the second argument.
        MEND                 ; Macro end
```

A macro that uses a condition code parameter:

```
        AREA    codx, CODE, READONLY

; macro definition
        MACRO
        Return$cond
        [ {ARCHITECTURE} <> "4"
          BX$cond lr
          |
          MOV$cond pc,lr
        ]
        MEND
; macro invocation
fun     PROC
        CMP      r0,#0
        MOVEQ    r0,#1
        ReturnEQ
        MOV      r0,#0
        Return
        ENDP
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
        END
```

Related information

MEXIT directive on page 1020
GBLA, GBLL, and GBLS directives on page 1006
LCLA, LCLL, and LCLS directives on page 1014
Use of macros on page 925
Assembly time substitution of variables on page 949

6.6.53 MAP directive

The `MAP` directive sets the origin of a storage map to a specified address.

Syntax

```
MAP <expr>{,<base-register>}
```

where:

`<expr>`

is a numeric or PC-relative expression:

- If `<base-register>` is not specified, `<expr>` evaluates to the address where the storage map starts. The storage map location counter is set to this address.
- If `<expr>` is PC-relative, you must have defined the label before you use it in the map. The map requires the definition of the label during the first pass of the assembler.

`<base-register>`

specifies a register. If `<base-register>` is specified, the address where the storage map starts is the sum of `<expr>`, and the value in `<base-register>` at runtime.

Usage

Use the `MAP` directive in combination with the `FIELD` directive to describe a storage map.

Specify `<base-register>` to define register-relative labels. The base register becomes implicit in all labels defined by following `FIELD` directives, until the next `MAP` directive. The register-relative labels can be used in load and store instructions.

The `MAP` directive can be used any number of times to define multiple storage maps.

The storage-map location counter, `{VAR}`, is set to the same address as that specified by the `MAP` directive. The `{VAR}` counter is set to zero before the first `MAP` directive is used.

`^` is a synonym for `MAP`.

Examples

```
    MAP     0,r9
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
    MAP     0xff,r9
```

Related information

FIELD directive on page 996
Directives that can be omitted in pass 2 of the assembler on page 969
How the assembler works

6.6.54 MEXIT directive

The `MEXIT` directive exits a macro definition before the end.

Usage

Use `MEXIT` when you require an exit from within the body of a macro. Any unclosed `WHILE...WEND` loops or `IF...ENDIF` conditions within the body of the macro are closed by the assembler before the macro is exited.

Example

```
        MACRO

$abc    example abc     $param1,$param2
        ; code
        WHILE condition1
            ; code
            IF condition2
                ; code
                MEXIT
            ELSE
                ; code
            ENDIF
        WEND
        ; code
        MEND
```

Related information

MACRO and MEND directives on page 1016

6.6.55 NOFP directive

The `NOFP` directive ensures that there are no floating-point instructions in an assembly language source file.

Syntax

```
NOFP
```

Usage

Use `NOFP` to ensure that no floating-point instructions are used in situations where there is no support for floating-point instructions either in software or in target hardware.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If a floating-point instruction occurs after the `NOFP` directive, an `Unknown opcode` error is generated and the assembly fails.

If a `NOFP` directive occurs after a floating-point instruction, the assembler generates the error:

```
Too late to ban floating-point instructions
```

and the assembly fails.

6.6.56 OPT directive

The `OPT` directive sets listing options from within the source code.

Syntax

```
OPT <n>
```

where:

<n>
is the `OPT` directive setting. The following table lists the valid settings:

Table 6-26: OPT directive settings

OPT n	Effect
1	Turns on normal listing.
2	Turns off normal listing.
4	Page throw. Issues an immediate form feed and starts a new page.
8	Resets the line number counter to zero.
16	Turns on listing for `SET`, `GBL` and `LCL` directives.
32	Turns off listing for `SET`, `GBL` and `LCL` directives.
64	Turns on listing of macro expansions.
128	Turns off listing of macro expansions.
256	Turns on listing of macro invocations.
512	Turns off listing of macro invocations.
1024	Turns on the first pass listing.
2048	Turns off the first pass listing.
4096	Turns on listing of conditional directives.
8192	Turns off listing of conditional directives.
16384	Turns on listing of `MEND` directives.
32768	Turns off listing of `MEND` directives.

Usage

Specify the `--list=` assembler option to turn on listing.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

By default the `--list=` option produces a normal listing that includes variable declarations, macro expansions, call-conditioned directives, and `MEND` directives. The listing is produced on the second pass only. Use the `OPT` directive to modify the default listing options from within your code.

You can use `OPT` to format code listings. For example, you can specify a new page before functions and sections.

Example

```
        AREA    Example, CODE, READONLY

start   ; code
        ; code
        BL      func1
        ; code
        OPT 4               ; places a page break before func1
func1   ; code
```

Related information

--list=file on page 885

6.6.57 QN, DN, and SN directives

The `QN`, `DN`, and `SN` directives define names for Advanced SIMD and floating-point registers.

Syntax

```
<name> <directive> <expr>{.<type>}{[<x>]}
```

Parameters

`<directive>`

`QN`, `DN`, or `SN`.

`<name>`

The name to be assigned to the extension register. `<name>` cannot be the same as any of the predefined names.

`<expr>`

Can be:

- An expression that evaluates to a number in the range:
  - 0-15 if you are using QN in A32/T32 Advanced SIMD code.
  - 0-31 otherwise.
- A predefined register name, or a register name that has already been defined in a previous directive.

Extended notation

The following are extended notation:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

<type>
Any Advanced SIMD or floating-point data type.

[<x>]
Only available for Advanced SIMD code. `[<x>]` is a scalar index into a register.

Operation

Use `QN` , `DN`, or `SN` to allocate convenient names to extension registers, to help you to remember what you use each one for.

The `QN` directive defines a name for a specified 128-bit extension register.

The `DN` directive defines a name for a specified 64-bit extension register.

The `SN` directive defines a name for a specified single-precision floating-point register.


Note

Avoid conflicting uses of the same register under different names.

You cannot specify a vector length in a `DN` or `SN` directive.

Example: DN and QN

```
energy  DN  6   ; defines energy as a symbol for

                ; floating-point double-precision register 6
mass    SN  16  ; defines mass as a symbol for
                ; floating-point single-precision register 16
```

Example: Extended notation

```
varA    DN      d1.U16
varB    DN      d2.U16
varC    DN      d3.U16
        VADD    varA,varB,varC       ; VADD.U16 d1,d2,d3
index   DN      d4.U16[0]
result  QN      q5.I32
        VMULL   result,varA,index    ; VMULL.U16 q5,d1,d4[0]
```

6.6.58 RELOC directive

The `RELOC` directive explicitly encodes an ELF relocation in an object file.

Syntax

```
RELOC <n>, <symbol>

RELOC <n>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

where:

<n>

must be an integer in the range 0 to 255 or one of the relocation names defined in the *Application Binary Interface for the Arm Architecture*.

<symbol>

can be any PC-relative label.

Usage

Use `RELOC <n>, <symbol>` to create a relocation with respect to the address labeled by `<symbol>`.

If used immediately after an A32 or T32 instruction, `RELOC` results in a relocation at that instruction. If used immediately after a `DCB`, `DCW`, or `DCD`, or any other data generating directive, `RELOC` results in a relocation at the start of the data. Any addend to be applied must be encoded in the instruction or in the data.

If the assembler has already emitted a relocation at that place, the relocation is updated with the details in the `RELOC` directive, for example:

```
    DCD     sym2 ; R_ARM_ABS32 to sym32
    RELOC   55   ; ... makes it R_ARM_ABS32_NOI
```

`RELOC` is faulted in all other cases, for example, after any non-data generating directive, `LTORG`, `ALIGN`, or as the first thing in an `AREA`.

Use `RELOC <n>` to create a relocation with respect to the anonymous symbol, that is, symbol 0 of the symbol table. If you use `RELOC <n>` without a preceding assembler generated relocation, the relocation is with respect to the anonymous symbol.

Examples

```
    IMPORT  impsym
    LDR     r0,[pc,#-8]
    RELOC   4, impsym
    DCD     0
    RELOC   2, sym
    DCD     0,1,2,3,4       ; the final word is relocated
    RELOC   38,sym2         ; R_ARM_TARGET1
    DCD     impsym
    RELOC   R_ARM_TARGET1   ; relocation code 38
```

Related information

Application Binary Interface for the Arm Architecture

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.59 REQUIRE directive

The `REQUIRE` directive specifies a dependency between sections.

Syntax

```
REQUIRE <label>
```

where:

`<label>`
is the name of the required label.

Usage

Use `REQUIRE` to ensure that a related section is included, even if it is not directly called. If the section containing the `REQUIRE` directive is included in a link, the linker also includes the section containing the definition of the specified label.

6.6.60 REQUIRE8 and PRESERVE8 directives

The `REQUIRE8` and `PRESERVE8` directives specify that the current file requires or preserves eight-byte alignment of the stack.


Note

This directive is required to support non-ABI conforming toolchains. It has no effect on AArch64 assembly and is not required when targeting AArch64.

Syntax

```
REQUIRE8 {\<bool>\}

PRESERVE8 {\<bool>\}
```

where:

`<bool>`
is an optional Boolean constant, either `{TRUE}` or `{FALSE}`.

Usage

Where required, if your code preserves eight-byte alignment of the stack, use `PRESERVE8` to set the `PRES8` build attribute on your file. If your code does not preserve eight-byte alignment of the stack, use `PRESERVE8 {FALSE}` to ensure that the `PRES8` build attribute is not set. Use `REQUIRE8` to set the `REQ8` build attribute. If there are multiple `REQUIRE8` or `PRESERVE8` directives in a file, the assembler uses the value of the last directive.

The linker checks that any code that requires eight-byte alignment of the stack is only called, directly or indirectly, by code that preserves eight-byte alignment of the stack.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If you omit both PRESERVE8 and PRESERVE8 {FALSE}, the assembler decides whether to set the PRES8 build attribute or not, by examining instructions that modify the SP. Arm recommends that you specify PRESERVE8 explicitly.


Note

You can enable a warning by using the --diag_warning 1546 option when invoking armasm.

This gives you warnings like:

```
"test.s", line 37: Warning: A1546W: Stack pointer update potentially
 breaks 8 byte stack alignment

       37 00000044         STMFD    sp!,{r2,r3,lr}
```

Examples

```
    REQUIRE8
    REQUIRE8     {TRUE}      ; equivalent to REQUIRE8
    REQUIRE8     {FALSE}     ; equivalent to absence of REQUIRE8
    PRESERVE8    {TRUE}      ; equivalent to PRESERVE8
    PRESERVE8    {FALSE}     ; NOT exactly equivalent to absence of PRESERVE8
```

Related information

--diag_warning=tag[,tag,...] (armasm) on page 875

6.6.61 RLIST directive

The RLIST (register list) directive gives a name to a set of general-purpose registers in A32/T32 code.

Syntax

```
<name> RLIST {\<list-of-registers>\}
```

where:

<name>

is the name to be given to the set of registers. <name> cannot be the same as any of the predefined names.

<list-of-registers>

is a comma-delimited list of register names and register ranges. The register list must be enclosed in braces.

Usage

Use RLIST to give a name to a set of registers to be transferred by the LDM or STM instructions.

LDM and STM always put the lowest physical register numbers at the lowest address in memory, regardless of the order they are supplied to the LDM or STM instruction. If you have defined your

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

own symbolic register names it can be less apparent that a register list is not in increasing register order.

Use the `--diag_warning 1206` assembler option to ensure that the registers in a register list are supplied in increasing register order. If registers are not supplied in increasing register order, a warning is issued.

Example

```
Context RLIST   {r0-r6,r8,r10-r12,pc}
```

6.6.62 RN directive

The RN directive defines a name for a specified register.

Syntax

```
<name> RN <expr>
```

where:

<name>

is the name to be assigned to the register. <name> cannot be the same as any of the predefined names.

<expr>

evaluates to a register number from 0 to 15.

Usage

Use RN to allocate convenient names to registers, to help you to remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

Examples

```
regname     RN  11  ; defines regname for register 11
sqr4        RN  r6  ; defines sqr4 for register 6
```

6.6.63 ROUT directive

The ROUT directive marks the boundaries of the scope of numeric local labels.

Syntax

```
{\<name>\} ROUT
```

where:

<name>

is the name to be assigned to the scope.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Usage

Use the ROUT directive to limit the scope of numeric local labels. This makes it easier for you to avoid referring to a wrong label by accident. The scope of numeric local labels is the whole area if there are no ROUT directives in it.

Use the <name> option to ensure that each reference is to the correct numeric local label. If the name of a label or a reference to a label does not match the preceding ROUT directive, the assembler generates an error message and the assembly fails.

Example

```
            ; code

routineA    ROUT            ; ROUT is not necessarily a routine
            ; code
3routineA   ; code          ; this label is checked
            ; code
            BEQ     %4routineA   ; this reference is checked
            ; code
            BGE     %3      ; refers to 3 above, but not checked
            ; code
4routineA   ; code          ; this label is checked
            ; code
otherstuff  ROUT            ; start of next scope
```

Related information

AREA directive on page 974
Numeric local labels on page 953

6.6.64 SETA, SETL, and SETS directives

The SETA, SETL, and SETS directives set the value of a local or global variable.

Syntax

```
<variable> <setx> <expr>
```

where:

<variable>
is the name of a variable declared by a GBLA, GBLL, GBLS, LCLA, LCLL, or LCLS directive.

<setx>
is one of SETA, SETL, or SETS

<expr>
is an expression that is:

- Numeric, for SETA.
- Logical, for SETL.
- String, for SETS.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Usage

The SETA directive sets the value of a local or global arithmetic variable.

The SETL directive sets the value of a local or global logical variable.

The SETS directive sets the value of a local or global string variable.

You must declare <variable> using a global or local declaration directive before using one of these directives.

You can also predefine variable names on the command line.

Restrictions

The value you can specify using a SETA directive is limited to 32 bits. If you exceed this limit, the assembler reports an error. A possible workaround in A64 code is to use an EQU directive instead of SETA, although EQU defines a constant, whereas GBLA and SETA define a variable.

For example, replace the following code:

```
                GBLA    MyAddress

MyAddress       SETA    0x0000008000000000
```

with:

```
MyAddress       EQU     0x0000008000000000
```

Examples

```
                GBLA    VersionNumber

VersionNumber   SETA    21
                GBLL    Debug
Debug           SETL    {TRUE}
                GBLS    VersionString
VersionString   SETS    "Version 1.0"
```

Related information

GBLA, GBLL, and GBLS directives on page 1006
LCLA, LCLL, and LCLS directives on page 1014
--predefine "directive" on page 891
String expressions on page 955
Numeric expressions on page 956
Logical expressions on page 959

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.65 SPACE and FILL directives

The SPACE directive reserves a zeroed block of memory. The FILL directive reserves a block of memory to fill with a given value.

Syntax

```
{\<label>\} SPACE <expr>

{\<label>\} FILL <expr>{,<value>{,<valuesize>}}
```

where:

<label>
is an optional label.

<expr>
evaluates to the number of bytes to fill or zero.

<value>
evaluates to the value to fill the reserved bytes with. <value> is optional and if omitted, it is 0. <value> must be 0 in a NOINIT area.

<valuesize>
is the size, in bytes, of <value>. It can be any of 1, 2, or 4. <valuesize> is optional and if omitted, it is 1.

Usage

Use the ALIGN directive to align any code following a SPACE or FILL directive.

% is a synonym for SPACE.

Example

```
        AREA    MyData, DATA, READWRITE

data1   SPACE   255       ; defines 255 bytes of zeroed store
data2   FILL    50,0xAB,1 ; defines 50 bytes containing 0xAB
```

Related information

ALIGN directive on page 971
DCB directive on page 984
DCD and DCDU directives on page 984
DCQ and DCQU directives on page 989
DCW and DCWU directives on page 990
Numeric expressions on page 956

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

6.6.66 THUMB directive

The THUMB directive instructs the assembler to interpret subsequent instructions as T32 instructions, using the UAL syntax.


Note

Not supported for AArch64 state.

Syntax

```
THUMB
```

Usage

In files that contain code using different instruction sets, the THUMB directive must precede T32 code written in UAL syntax.

If necessary, this directive also inserts one byte of padding to align to the next halfword boundary.

This directive does not assemble to any instructions. It also does not change the state. It only instructs armasm to assemble T32 instructions as appropriate, and inserts padding if necessary.

Example

This example shows how you can use ARM and THUMB directives to switch state and assemble both A32 and T32 instructions in a single area.

```
        AREA ToT32, CODE, READONLY      ; Name this block of code

        ENTRY                           ; Mark first instruction to execute
        ARM                             ; Subsequent instructions are A32
start
        ADR     r0, into_t32 + 1        ; Processor starts in A32 state
        BX      r0                      ; Inline switch to T32 state
        THUMB                           ; Subsequent instructions are T32
into_t32
        MOVS    r0, #10                 ; New-style T32 instructions
```

Related information

ARM or CODE32 directive on page 978
CODE16 directive on page 981

6.6.67 TTL and SUBT directives

The TTL directive inserts a title at the start of each page of a listing file. The SUBT directive places a subtitle on the pages of a listing file.

Syntax

```
TTL <title>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
SUBT <subtitle>
```

where:

<title>
is the title.

<subtitle>
is the subtitle.

Usage

Use the TTL directive to place a title at the top of each page of a listing file. If you want the title to appear on the first page, the TTL directive must be on the first line of the source file.

Use additional TTL directives to change the title. Each new TTL directive takes effect from the top of the next page.

Use SUBT to place a subtitle at the top of each page of a listing file. Subtitles appear in the line below the titles. If you want the subtitle to appear on the first page, the SUBT directive must be on the first line of the source file.

Use additional SUBT directives to change subtitles. Each new SUBT directive takes effect from the top of the next page.

Examples

```
    TTL   First Title    ; places title on first and subsequent pages of listing
 file.
    SUBT  First Subtitle ; places subtitle on second and subsequent pages of listing
 file.
```

6.6.68 WHILE and WEND directives

The WHILE directive starts a sequence of instructions or directives that are to be assembled repeatedly. The sequence is terminated with a WEND directive.

Syntax

```
WHILE <logical-expression>
  code
WEND
```

Parameters

<logical-expression>
An expression that can evaluate to either {TRUE} or {FALSE}.

Operation

Use the WHILE directive, together with the WEND directive, to assemble a sequence of instructions a number of times. The number of repetitions can be zero.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

You can use IF...ENDIF conditions within WHILE...WEND loops.

WHILE...WEND loops can be nested.

Example

```
        GBLA count                  ; declare local variable

count   SETA    1                   ; you are not restricted to
        WHILE   count <= 4          ; such simple conditions
count   SETA    count+1             ; In this case, this code is
            ; code                  ; executed four times
            ; code                  ;
        WEND
```

Related information

Logical expressions on page 959
About armasm assembly language control directives on page 967

6.6.69 WN and XN

The WN, and XN directives define names for registers in A64 code.

The WN directive defines a name for a specified 32-bit register.

The XN directive defines a name for a specified 64-bit register.

Syntax

```
<name> <directive> <expr>
```

where:

<name>
is the name to be assigned to the register. <name> cannot be the same as any of the predefined names.

<directive>
is WN or XN.

<expr>
evaluates to a register number from 0 to 30.

Usage

Use WN and XN to allocate convenient names to registers in A64 code, to help you to remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

Examples

```
sqr4        WN w16  ; defines sqr4 for register w16
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
regname     XN 21  ; defines regname for register x21
```

6.7 armasm-Specific A32 and T32 Instruction Set Features

armasm provides additional support for the Arm instruction set and has specific behavior for some instructions.

armasm pseudo-instructions

The following table summarizes the pseudo-instructions available in the A32 and T32 instruction sets:

Table 6-27: Summary of pseudo-instructions

Mnemonic	Brief description	See
ADRL pseudo-instruction	Load program or register-relative address (medium range)	ADRL pseudo-instruction
CPY pseudo-instruction	Copy	CPY pseudo-instruction
LDR pseudo-instruction	Load Register	LDR pseudo-instruction
MOV32 pseudo-instruction	Move 32-bit immediate to register	MOV32 pseudo-instruction
NEG pseudo-instruction	Negate	NEG pseudo-instruction
UND pseudo-instruction	Generate an architecturally undefined instruction.	UND pseudo-instruction

Instructions for which armasm has specific behavior

The following table lists instructions for which armasm has specific behavior:

Table 6-28: Summary of instructions with armasm-specific behavior

Mnemonic	Brief description	See
CSDB instruction	Consumption of Speculative Data Barrier instruction.	armasm support for the CSDB instruction
IT instruction	If-Then instruction	IT instruction (A32)
SEVL instruction	Set Event Locally instruction	SEVL instruction (A32)

6.7.1 armasm support for the CSDB instruction

For conditional CSDB instructions that specify a condition <c> other than AL in A32, and for any condition <c> used inside an IT block in T32, then armasm rejects conditional CSDB instructions, outputs an error message, and aborts.

For example:

- For A32 code:

```
"test2.s", line 4: Error: A1895E: The specified condition results in
 UNPREDICTABLE behaviour
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
          4 00000000   CSDBEQ
```

- For T32 code:

```
"test2.s", line 8: Error: A1603E: This instruction inside IT block has
 UNPREDICTABLE results

          8 00000006   CSDBEQ
```

You can relax this behavior by using:

- The `--diag-suppress=1895` option for A32 code.
- The `--diag-suppress=1603` option for T32 code.

You can also use the `--unsafe` option with these options. However, this option disables many correctness checks.

Related information

--diag_suppress=tag[,tag,...] (armasm) on page 874
--unsafe on page 894
CSDB instruction

6.7.2 ADRL pseudo-instruction

Load a PC-relative or register-relative address into a register.

Syntax

```
ADRL cond <Rd>,<label>
```

Parameters

<cond>
An optional condition code.

<Rd>
The register to load.

<label>
A PC-relative or register-relative expression.

Restrictions

`ADRL` is not available in Arm®v6-M and Armv8-M Baseline.

Operation

`ADRL` always assembles to two 32-bit instructions. Even if the address can be reached in a single instruction, a second, redundant instruction is produced.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

If the assembler cannot construct the address in two instructions, it generates an error message and the assembly fails. You can use the `LDR` pseudo-instruction for loading a wider range of addresses.

`ADRL` is similar to the `ADR` instruction, except `ADRL` can load a wider range of addresses because it generates two data processing instructions.

`ADRL` produces position independent code, because the address is PC-relative or register-relative.

If `<label>` is PC-relative, it must evaluate to an address in the same assembler area as the `ADRL` pseudo-instruction.

If you use `ADRL` to generate a target for a `BX` or `BLX` instruction, it is your responsibility to set the T32 bit (bit 0) of the address if the target contains T32 instructions.

Architectures and range

The available range depends on the instruction set in use:

A32

The range of the instruction is any value that can be generated by two `ADD` or two `SUB` instructions. That is, any value that can be produced by the addition of two values, each of which is 8 bits rotated right by any even number of bits within a 32-bit word.

T32, 32-bit encoding

±1MB bytes to a byte, halfword, or word-aligned address.

T32, 16-bit encoding

`ADRL` is not available.

The given range is relative to a point four bytes (in T32 code) or two words (in A32 code) after the address of the current instruction.

Related information

Register-relative and PC-relative expressions on page 950
Load immediate values on page 907
LDR pseudo-instruction on page 1038
Arm Architecture Reference Manuals

6.7.3 CPY pseudo-instruction

Copy a value from one register to another.

Syntax

```
CPY {<cond>} <Rd>, <Rm>
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Parameters

<cond>

An optional condition code.

<Rd>

The destination register.

<Rm>

The register holding the value to be copied.

Operation

The `CPY` pseudo-instruction copies a value from one register to another, without changing the condition flags.

`CPY <Rd>, <Rm>` assembles to `MOV <Rd>, <Rm>`.

Architectures

This pseudo-instruction is available in A32 code and in T32 code.

Register restrictions

Using SP or PC for both `<Rd>` and `<Rm>` is deprecated.

Condition flags

This instruction does not change the condition flags.

Related information

MOV

6.7.4 IT instruction (A32)

The `IT` (If-Then) instruction makes a single instruction following `IT` conditional. The `IT` instruction and the following instruction are referred to as an IT block. The conditional instruction must be from a restricted set of 16-bit instructions.

The following instructions are not permitted in an IT block:

- `IT`.
- `CBZ` and `CBNZ`.
- `TBB` and `TBH`.
- `CPS`, `CPSID` and `CPSIE`.
- `SETEND`.



Note

`armasm` displays a diagnostic message when any of these instructions are used in an IT block.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Related information

IT

6.7.5 LDR pseudo-instruction

Load a register with either a 32-bit immediate value or an address.


Note

This information is for the LDR pseudo-instruction only, and not the LDR instruction.

Syntax

```
LDR{<cond>}{.W} <Rt>, =<expr>
LDR{<cond>}{.W} <Rt>, =<label_expr>
```

Parameters

<cond>
An optional condition code.

.W
An optional instruction width specifier.

<Rt>
The register to be loaded.

<expr>
Evaluates to a numeric value.

<label_expr>
A PC-relative or external expression of an address in the form of a label plus or minus a numeric value.

Operation

When using the LDR pseudo-instruction:

- If the value of <expr> can be loaded with a valid MOV or MVN instruction, the assembler uses that instruction.
- If a valid MOV or MVN instruction cannot be used, or if the label_expr syntax is used, the assembler places the constant in a literal pool and generates a PC-relative LDR instruction that reads the constant from the literal pool.


Note

  - An address loaded in this way is fixed at link time, so the code is not position independent.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The address holding the constant remains valid regardless of where the linker places the ELF section containing the LDR instruction.

The assembler places the value of <label_expr> in a literal pool and generates a PC-relative LDR instruction that loads the value from the literal pool.

If <label_expr> is an external expression, or is not contained in the current section, the assembler places a linker relocation directive in the object file. The linker generates the address at link time.

If <label_expr> is either a named or numeric local label, the assembler places a linker relocation directive in the object file and generates a symbol for that local label. The address is generated at link time. If the local label references T32 code, the T32 bit (bit 0) of the address is set.

The offset from the PC to the value in the literal pool must be less than ±4KB (in an A32 or 32-bit T32 encoding) or in the range 0 to +1KB (16-bit T32 encoding). You are responsible for ensuring that there is a literal pool within range.

If the label referenced is in T32 code, the LDR pseudo-instruction sets the T32 bit (bit 0) of <label_expr>.


Note

In *RealView Compilation Tools* (RVCT) v2.2, the T32 bit of the address was not set. If you have code that relies on this behavior, use the command-line option --untyped_local_labels to force the assembler not to set the T32 bit when referencing labels in T32 code.

LDR in T32 code

You can use the .W width specifier to force LDR to generate a 32-bit instruction in T32 code. LDR.W always generates a 32-bit instruction, even if the immediate value could be loaded in a 16-bit MOV, or there is a literal pool within reach of a 16-bit PC-relative load.

If the value to be loaded is not known in the first pass of the assembler, LDR without .W generates a 16-bit instruction in T32 code, even if that results in a 16-bit PC-relative load for a value that could be generated in a 32-bit MOV or MVN instruction. However, if the value is known in the first pass, and it can be generated using a 32-bit MOV or MVN instruction, the MOV or MVN instruction is used.

In UAL syntax, the LDR pseudo-instruction never generates a 16-bit flag-setting MOV instruction. Use the --diag_warning 1727 assembler command-line option to check when a 16-bit instruction could have been used.

You can use the MOV32 pseudo-instruction for generating immediate values or addresses without loading from a literal pool.

Examples

```
LDR     r3,=0xff0     ; loads 0xff0 into R3
                      ; =>  MOV.W r3,#0xff0
LDR     r1,=0xfff     ; loads 0xfff into R1
                      ; =>  LDR r1,[pc,offset_to_litpool]
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
                        ;     ...
                        ;     litpool DCD 0xfff
    LDR     r2,=place   ; loads the address of
                        ; place into R2
                        ; =>  LDR r2,[pc,offset_to_litpool]
                        ;     ...
                        ;     litpool DCD place
```

Related information

--untyped_local_labels on page 895
Numeric constants on page 948
Register-relative and PC-relative expressions on page 950
Numeric local labels on page 953
MOV32 pseudo-instruction on page 1040

6.7.6 MOV32 pseudo-instruction

Load a register with either a 32-bit immediate value or any address.

Syntax

```
MOV32{<cond>} <Rd>, <expr>
```

Parameters

<cond>

An optional condition code.

<Rd>

The register to be loaded. `<Rd>` must not be SP or PC.

<expr>

Can be any one of the following:

<symbol>

A label in this or another program area.

<#constant>

Any 32-bit immediate value.

{<symbol> + <constant>}

A label plus a 32-bit immediate value.

Operation

`MOV32` always generates two 32-bit instructions, a `MOV`, `MOVT` pair. This enables you to load any 32-bit immediate, or to access the whole 32-bit address space.

The main purposes of the `MOV32` pseudo-instruction are:

- To generate literal constants when an immediate value cannot be generated in a single instruction.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- To load a PC-relative or external address into a register. The address remains valid regardless of where the linker places the ELF section containing the `MOV32`.

> **Note**
>
> An address loaded in this way is fixed at link time, so the code is not position independent.

`MOV32` sets the T32 bit (bit 0) of the address if the label referenced is in T32 code.

Architectures

This pseudo-instruction is available in A32 and T32.

Examples

```
    MOV32 r3, #0xABCDEF12  ; loads 0xABCDEF12 into R3

    MOV32 r1, Trigger+12   ; loads the address that is 12 bytes
                           ; higher than the address Trigger into R1
```

Related information

Condition code suffixes

6.7.7 NEG pseudo-instruction

Negate the value in a register.

Syntax

```
NEG{<cond>} <Rd>, <Rm>
```

Parameters

<cond>

An optional condition code.

<Rd>

The destination register.

<Rm>

The register containing the value that is subtracted from zero.

Operation

The `NEG` pseudo-instruction negates the value in one register and stores the result in a second register.

`NEG`*cond*`<Rd>, <Rm>` assembles to `RSBS`*cond*`<Rd>, <Rm>, #0`.

Architectures

The 32-bit encoding of this pseudo-instruction is available in A32 and T32.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

There is no 16-bit encoding of this pseudo-instruction available T32.

Register restrictions

In A32 instructions, using SP or PC for `<Rd>` or `<Rm>` is deprecated. In T32 instructions, you cannot use SP or PC for `<Rd>` or `<Rm>`.

Condition flags

This pseudo-instruction updates the condition flags, based on the result.

Related information

ADD

6.7.8 SEVL instruction (A32)

The `SEVL` (Set Event Locally) instruction is a hint instruction and is supported only in Arm®v8. It is optional whether `SEVL` is implemented or not. If `SEVL` is not implemented, it executes as a `NOP`. `armasm` produces a diagnostic message if the instruction executes as a `NOP` on the target.

Related information

SEVL

6.7.9 UND pseudo-instruction

Generate an architecturally **UNDEFINED** instruction.

Syntax

```
UND{<cond>}{.W} {#<expr>}
```

Parameters

`<cond>`

An optional condition code.

`.W`

An optional instruction width specifier.

`<expr>`

Evaluates to a numeric value. The following table shows the range and encoding of `<expr>` in the instruction, where Y shows the locations of the bits that encode for `<expr>` and V is the 4 bits that encode for the condition code.

If `<expr>` is omitted, the value 0 is used.

Table 6-29: Range and encoding of expr

Instruction	Encoding	Number of bits for `<expr>`	Range
A32	0xV7FYYYFY	16	0-65535

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Instruction	Encoding	Number of bits for <expr>	Range
T32 32-bit encoding	0xF7FYAYFY	12	0-4095
T32 16-bit encoding	0xDEYY	8	0-255

Operation

An attempt to execute an undefined instruction causes the Undefined instruction exception. Architecturally **UNDEFINED** instructions are expected to remain undefined.

UND in T32 code

You can use the `.W` width specifier to force `UND` to generate a 32-bit instruction in T32 code. `UND.W` always generates a 32-bit instruction, even if `<expr>` is in the range 0-255.

Disassembly

The encodings that this pseudo-instruction produces disassemble to `DCI`.

Related information

Condition code suffixes

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Appendix A Supporting Information Appendix

This appendix contains supporting information for this document.

A.1 Standard C Implementation Definition

Provides information required by the ISO C standard for conforming C implementations.

A.1.1 Implementation definition (ISO C Standard)

Appendix J of the ISO C standard (ISO/IEC 9899:2011 (E)) contains information about portability issues. Sub-clause J3 lists the behavior that each implementation must document.

The following topics correspond to the relevant sections of sub-clause J3. They describe aspects of the Arm C Compiler and C library, not defined by the ISO C standard, that are implementation-defined. Whenever the implementation-defined behavior of the Arm C compiler or the C library can be altered and tailored to the execution environment by reimplementing certain functions, that behavior is described as "depends on the environment".

Related information

Translation on page 1044
Translation limits on page 1045
Environment on page 1046
Identifiers on page 1048
Characters on page 1048
Integers on page 1050
Floating-point on page 1051
Arrays and pointers on page 1052
Hints on page 1053
Structures, unions, enumerations, and bitfields on page 1053
Qualifiers on page 1054
Preprocessing directives (ISO C Standard) on page 1054
Library functions on page 1056
Architecture on page 1061

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A.1.2 Translation

Describes implementation-defined aspects of the Arm C compiler and C library relating to translation, as required by the ISO C standard.

How a diagnostic is identified (3.10, 5.1.1.3).

Diagnostic messages that the compiler produces are of the form:

```
<source-file>:<line-number>:<char-number>:<description> [<diagnostic-flag>]
```

Here:

<description>

Is a text description of the error.

<diagnostic-flag>

Is an optional diagnostic flag of the form `-W<flag>`, only for messages that can be suppressed.

Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

Each nonempty sequence of white-space characters, other than new-line, is replaced by one space character.

A.1.3 Translation limits

Describes implementation-defined aspects of the Arm C compiler and C library relating to translation, as required by the ISO C standard.

Section *5.2.4.1 Translation limits* of the ISO/IEC 9899:2011 standard requires minimum translation limits that a conforming compiler must accept. The following table gives a summary of these limits. In this table, a limit of `<memory>` indicates that Arm® Compiler for Embedded 6 imposes no limit, other than the limit imposed by available memory.

The compiler reports one of the following errors for a translation unit that is larger than the maximum acceptable size of approximately 2GB:

```
file '<filename>' is too large for Clang to process
translation unit is too large for Clang to process: ran out of source locations
```

Table A-1: Translation limits

Description	Translation limit
Nesting levels of block.	256 (can be increased using the `-fbracket-depth` option.)
Nesting levels of conditional inclusion.	`<memory>`
Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or void type in a declaration.	`<memory>`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Description	Translation limit
Nesting levels of parenthesized declarators within a full declarator.	256 (can be increased using the `-fbracket-depth` option.)
Nesting levels of parenthesized expressions within a full expression.	256 (can be increased using the `-fbracket-depth` option.)
Significant initial characters in an internal identifier or a macro name.	`<memory>`
Significant initial characters in an external identifier.	`<memory>`
External identifiers in one translation unit.	`<memory>`
Identifiers with block scope declared in one block.	`<memory>`
Macro identifiers simultaneously defined in one preprocessing translation unit.	`<memory>`
Parameters in one function definition.	`<memory>`
Arguments in one function call.	`<memory>`
Parameters in one macro definition.	`<memory>`
Arguments in one macro invocation.	`<memory>`
Characters in a logical source line.	`<memory>`
Characters in a string literal.	`<memory>`
Bytes in an object.	`<SIZE_MAX>`
Nesting levels for `#include` files.	`<memory>`
Case labels for a switch statement.	`<memory>`
Members in a single structure or union.	`<memory>`
Enumeration constants in a single enumeration.	`<memory>`
Levels of nested structure or union definitions in a single struct-declaration-list.	256 (can be increased using the `-fbracket-depth` option.)

Related information

-fbracket-depth=N on page 51

A.1.4 Environment

Describes implementation-defined aspects of the Arm C compiler and C library relating to environment, as required by the ISO C standard.

The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

The compiler interprets the physical source file multibyte characters as UTF-8.

The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

When linking with microlib, the function `main()` must be declared to take no arguments and must not return.

The effect of program termination in a freestanding environment (5.1.2.1).

The function `exit()` is not supported by microlib and the function `main()` must not return.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

An alternative manner in which the main function can be defined (5.1.2.2.1).

The main function can be defined in one of the following forms:

```
int main(void)
int main()
int main(int)
int main(int, char **)
int main(int,char **, char **)
```

The values given to the strings pointed to by the `argv` argument to main (5.1.2.2.1).

In the generic Arm library 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.

What constitutes an interactive device (5.1.2.3).

What constitutes an interactive device depends on the environment and the `_sys_istty` function. The standard I/O streams `stdin`, `stdout`, and `stderr` are assumed to be interactive devices. They are line-buffered at program startup, regardless of what `_sys_istty` reports for them. An exception is if they have been redirected on the command line.

Whether a program can have more than one thread of execution in a freestanding environment (5.1.2.4).

Depends on the environment. The microlib C library is not thread-safe.

The set of signals, their semantics, and their default handling (7.14).

The `<signal.h>` header defines the following signals:

Signal	Value	Semantics
SIGABRT	1	Abnormal termination
SIGFPE	2	Arithmetic exception
SIGILL	3	Illegal instruction execution
SIGINT	4	Interactive attention signal
SIGSEGV	5	Bad memory access
SIGTERM	6	Termination request
SIGSTAK	7	Stack overflow (obsolete)
SIGRTRED	8	Run-time redirection error
SIGRTMEM	9	Run-time memory error
SIGUSR1	10	Available for the user
SIGUSR2	11	Available for the user
SIGPVFN	12	Pure virtual function called
SIGCPPL	13	Not normally used
SIGOUTOFHEAP	14	`::operator new` or `::operator new[]` cannot allocate memory

The default handling of all recognized signals is to print a diagnostic message and call `exit()`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).

No signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` correspond to a computational exception.

Signals for which the equivalent of `signal(sig, SIG_IGN);` is executed at program startup (7.14.1.1).

No signals are ignored at program startup.

The set of environment names and the method for altering the environment list used by the getenv function (7.22.4.6).

The default implementation returns `NULL`, indicating that no environment information is available.

The manner of execution of the string by the system function (7.22.4.8).

Depends on the environment. The default implementation of the function uses semihosting.

A.1.5 Identifiers

Describes implementation-defined aspects of the Arm C compiler and C library relating to identifiers, as required by the ISO C standard.

Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

Multibyte characters, whose UTF-8 decoded value falls within one of the ranges in Appendix D of ISO/IEC 9899:2011 are allowed in identifiers and correspond to the universal character name with the short identifier (as specified by ISO/IEC 10646) having the same numeric value.

The dollar character `$` is allowed in identifiers.

The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

There is no limit on the number of significant initial characters in an identifier.

A.1.6 Characters

Describes implementation-defined aspects of the Arm C compiler and C library relating to characters, as required by the ISO C standard.

The number of bits in a byte (3.6).

The number of bits in a byte is 8.

The values of the members of the execution character set (5.2.1).

The values of the members of the execution character set are all the code points defined by ISO/IEC 10646.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

Character escape sequences have the following values in the execution character set:

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

The value of a `char` object into which has been stored any character other than a member of the basic execution character set (6.2.5).

The value of a `char` object into which has been stored any character other than a member of the basic execution character set is the least significant 8 bits of that character, interpreted as `unsigned`.

Which of `signed char` or `unsigned char` has the same range, representation, and behavior as plain char (6.2.5, 6.3.1.1).

Data items of type `char` are `unsigned` by default. The type `unsigned char` has the same range, representation, and behavior as `char`.



You must take care when mixing translation units that are compiled with and without the [COMMUNITY] `-fsigned-char` 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 supplied with the Arm compilation tools. See Support level definitions.

The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

The execution character set is identical to the source character set.

The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

In C all character constants have 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.

The value of a wide-character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

If a wide-character constant contains more than one multibyte character, the compiler reports an error.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The current locale used to convert a wide-character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide-character code (6.4.4.4).

Mapping of wide-character constants to the corresponding wide-character code is locale independent.

Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence (6.4.5).

Differently prefixed wide string literal tokens cannot be concatenated.

The current locale used to convert a wide string literal into corresponding wide-character codes (6.4.5).

Mapping of the wide-characters in a wide string literal into the corresponding wide-character codes is locale independent.

The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

The compiler does not check if the value of a multibyte character or an escape sequence is a valid ISO/IEC 10646 code point. Such a value is encoded like the values of the valid members of the execution character set, according to the kind of the string literal (character or wide-character).

The encoding of any of `wchar_t`, `char16_t`, and `char32_t` where the corresponding standard encoding macro (`__STDC_ISO_10646__`, `__STDC_UTF_16__`, or `__STDC_UTF_32__`) is not defined (6.10.8.2).

The symbol `__STDC_ISO_10646__` is not defined. Nevertheless every character in the Unicode required set, when stored in an object of type `wchar_t`, has the same value as the short identifier of that character.

The symbols `__STDC_UTF_16__` and `__STDC_UTF_32__` are defined.

A.1.7 Integers

Describes implementation-defined aspects of the Arm C compiler and C library relating to integers, as required by the ISO C standard.

Any extended integer types that exist in the implementation (6.2.5).

No extended integer types exist in the implementation.

Whether signed integer types are represented using sign and magnitude, two's complement, or ones' complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).

Signed integer types are represented using two's complement with no padding bits. There is no extraordinary value.

The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

No extended integer types exist in the implementation.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

When converting an integer to a N-bit wide signed integer type and the value cannot be represented in the destination type, the representation of the source operand is truncated to N-bits and the resulting bit patters is interpreted a value of the destination type. No signal is raised.

The results of some bitwise operations on signed integers (6.5).

In the bitwise right shift `E1 >> E2`, if `E1` has a signed type and a negative value, the value of the result is the integral part of the quotient of `E1 / 2^E2`, except that shifting the value -1 yields result -1.

A.1.8 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.

The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (5.2.4.2.2).

Floating-point quantities are stored in IEEE format:

- `float` values are represented by IEEE single-precision values
- `double` values are represented by IEEE double-precision values.
- `long double` values in AArch32 are represented by IEEE double-precision values.
- `long double` values in AArch64 are represented by IEEE quadruple-precision values.


Note

The `long double` data type is not supported for AArch64 state because of limitations in the current Arm C library.

The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>` (5.2.4.2.2).

The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>` is unknown.

The rounding behaviors characterized by non-standard values of `FLT_ROUNDS` (5.2.4.2.2).

Arm® Compiler for Embedded does not define non-standard values for `FLT_ROUNDS`.

The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD` (5.2.4.2.2).

Arm Compiler for Embedded does not define non-standard values for `FLT_EVAL_METHOD`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

The direction of rounding when an integer is converted to a floating point number is "round to nearest, ties to even".

The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

When a floating-point number is converted to a different floating-point type and the value is within the range of the destination type, but cannot be represented exactly, the rounding mode is "round to nearest, ties to even", by default.

How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

When a floating-point literal is converted to a floating-point value, the rounding mode is "round to nearest, ties to even".

Whether and how floating expressions are contracted when not disallowed by the `FP_CONTRACT` pragma (6.5).

When not disallowed, expressions are contracted. This may be wherever possible across C statements if `-ffp-contract=fast`, or only in the same C statement if `-ffp-contract=on`. See -ffp-contract for more information.

The default state for the `FENV_ACCESS` pragma (7.6.1).

The default state of the `FENV_ACCESS` pragma is `OFF`. The state `ON` is not supported.

Additional floating-point exceptions, rounding classifications, and their macro names (7.6, 7.12), modes, environments, and the default state for the `FP_CONTRACT` pragma (7.12.2).

No additional floating-point exceptions, rounding classifications, modes, or environments are defined.

The `-ffp-contract` option sets the default value of the `FP_CONTRACT` pragma.

Related information

-ffp-contract on page 61
-ffp-mode on page 62
-ffast-math, -fno-fast-math on page 56
#pragma STDC on page 329
IEEE 754 arithmetic
IEEE 754 arithmetic and rounding

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A.1.9 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 result of converting a pointer to an integer or vice versa (6.3.2.3).

Converting a pointer to an integer type with smaller bit width discards the most significant bits of the pointer. Converting a pointer to an integer type with greater bit width zero-extends the pointer. Otherwise the bits of the representation are unchanged.

Converting an unsigned integer to pointer with a greater bit-width zero-extends the integer. Converting a signed integer to pointer with a greater bit-width sign-extends the integer. Otherwise the bits of the representation are unchanged.

The size of the result of subtracting two pointers to elements of the same array (6.5.6).

The size of the result of subtracting two pointers to elements of the same array is 4 bytes for AArch32 state, and 8 bytes for AArch64 state.

A.1.10 Hints

Describes implementation-defined aspects of the Arm C compiler and C library relating to registers, as required by the ISO C standard.

The extent to which suggestions made by using the register storage-class specifier are effective (6.7.1).

The register storage-class specifier is ignored as a means to control how fast the access to an object is. For example, an object might be allocated in register or allocated in memory regardless of whether it is declared with register storage-class.

The extent to which suggestions made by using the inline function specifier are effective (6.7.4).

The inline function specifier is ignored as a means to control how fast the calls to the function are made. For example, a function might be inlined or not regardless of whether it is declared inline.

A.1.11 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.

Whether a plain `int` bitfield is treated as a `signed int` bitfield or as an `unsigned int` bitfield (6.7.2, 6.7.2.1).

Plain `int` bitfields are signed.

Allowable bitfield types other than `_Bool`, `signed int`, and `unsigned int` (6.7.2.1).

Other integer types such as `char`, `short`, `long`, and `long long` (`signed` and `unsigned`), and enumeration types are allowed as bitfield types.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Whether atomic types are permitted for bitfields (6.7.2.1).

Atomic types are not permitted for bitfields.

Whether a bitfield can straddle a storage-unit boundary (6.7.2.1).

A bitfield cannot straddle a storage-unit boundary.

The order of allocation of bitfields within a unit (6.7.2.1).

Within a storage unit, successive bitfields are allocated from low-order bits towards high-order bits when compiling for little-endian, or from the high-order bits towards low-order bits when compiling for big-endian.

The alignment of non-bitfield members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

The non-bitfield members of structures of a scalar type are aligned to their size. The non-bitfield members of an aggregate type are aligned to the maximum of the alignments of each top-level member.

The integer type compatible with each enumerated type (6.7.2.2).

An enumerated type is compatible with `int` or `unsigned int`. If both the `signed` and the `unsigned` integer types can represent the values of the enumerators, the `unsigned` variant is chosen. If a value of an enumerator cannot be represented with `int` or `unsigned int`, then `long long` or `unsigned long long` is used.

A.1.12 Qualifiers

Describes implementation-defined aspects of the Arm C compiler and C library relating to qualifiers, as required by the ISO C standard.

What constitutes an access to an object that has volatile-qualified type (6.7.3).

Modifications of an object that has a volatile qualified type constitutes an access to that object. Value computation of an lvalue expression with a volatile qualified type constitutes an access to the corresponding object, even when the value is discarded.

A.1.13 Preprocessing directives (ISO C Standard)

Describes implementation-defined aspects of the Arm C compiler and C library relating to preprocessing directives, as required by the ISO C standard.

The locations within `#pragma` directives where header name preprocessing tokens are recognized (6.4, 6.4.7).

The compiler does not support pragmas that refer to headers.

How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

In both forms of the `#include` directive, the character sequences are mapped to external header names.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

The value of a character constant in conditional inclusion expression is the same as the value of the same constant in the execution character set.

Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

Single-character constants in conditional inclusion expressions have non-negative values.

The places that are searched for an included < > delimited header, and how the places are specified or the header is identified (6.10.2).

If the character sequence begins with the `/` character, it is interpreted as an absolute file path name.

Otherwise, the character sequence is interpreted as a file path, relative to one of the following directories:

- The sequence of the directories, given using the `-I` command-line option, in the command line order.
- The `include` subdirectory in the compiler installation directory.

How the named source file is searched for in an included " " delimited header (6.10.2).

If the character sequence begins with the `/` character, it is interpreted as an absolute file path name.

Otherwise, the character sequence interpreted as a file path, relative to the parent directory of the source file, which contains the `#include` directive.

The method by which preprocessing tokens (possibly resulting from macro expansion) in a `#include` directive are combined into a header name (6.10.2).

After macro replacement, the sequence of preprocessing tokens must be in one of the following two forms:

- A single string literal. The escapes in the string are not processed and adjacent string literals are not concatenated. Then the rules for double-quoted includes apply.
- A sequence of preprocessing tokens, starting with < ' and terminating with >. Sequences of whitespace characters, if any, are replaced by a single space. Then the rules for angle-bracketed includes apply.

The nesting limit for `#include` processing (6.10.2).

There is no limit to the nesting level of files included with `#include`.

Whether the `#` operator inserts a `\` character before the `\` character that begins a universal character name in a character constant or string literal (6.10.3.2).

A `\` character is inserted before the `\` character that begins a universal character name.

The behavior on each recognized non-standard C `#pragma` directive (6.10.6).

For the behavior of each non-standard C `#pragma` directive, see Compiler-specific Pragmas.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (6.10.8.1).

The date and time of the translation are always available on all supported platforms.

A.1.14 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.

Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).

Arm® Compiler for Embedded provides the Arm C Micro-library. For information about facilities, provided by this library, see The Arm C Micro-library in the *Arm C and C++ Libraries and Floating-Point Support User Guide*.

The format of the diagnostic printed by the assert macro (7.2.1.1).

The assert macro prints a diagnostic in the format:

```
*** assertion failed: <expression>, <filename>, <line number>
```

The representation of the floating-points status flags stored by the `fegetexceptflag` function (7.6.2.2).

The `fegetexceptflag` function stores the floating-point status flags as a bit set as follows:

- Bit 0 (0x01) is for the Invalid Operation exception.
- Bit 1 (0x02) is for the Divide by Zero exception.
- Bit 2 (0x04) is for the Overflow exception.
- Bit 3 (0x08) is for the Underflow exception.
- Bit 4 (0x10) is for the Inexact Result exception.

Whether the `feraiseexcept` function raises the Inexact floating-point exception in addition to the Overflow or Underflow floating-point exception (7.6.2.3).

The `feraiseexcept` function does not raise by itself the Inexact floating-point exception when it raises either an Overflow or Underflow exception.

Strings other than "C" and "" that can be passed as the second argument to the `setlocale` function (7.11.1.1).

What other strings can be passed as the second argument to the `setlocale` function depends on which `__use_X_ctype` symbol is imported (`__use_iso8859_ctype`, `__use_sjis_ctype`, or `__use_utf8_ctype`), and on user-defined locales.

The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 (7.12).

The types defined for `float_t` and `double_t` are float and double, respectively, for all the supported values of `FLT_EVAL_METHOD`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Domain errors for the mathematics functions, other than those required by this International Standard (7.12.1).

The following functions return additional domain errors under the specified conditions (the function name refers to all the variants of the function. For example, the `acos` entry applies to `acos`, `ascof`, and `acosl` functions):

Function	Condition	Return value	Error
`acos(x)`	`abs(x) > 1`	`NaN`	`EDOM`
`asin(x)`	`abs(x) > 1`	`NaN`	`EDOM`
`cos(x)`	`X == Inf`	`NaN`	`EDOM`
`sin(x)`	`x == Inf`	`NaN`	`EDOM`
`tan(x)`	`x == Inf`	`NaN`	`EDOM`
`atanh(x)`	`abs(x) == 1`	`Inf`	`ERANGE`
`ilogb(x)`	`x == 0.0`	`-INT_MAX`	`EDOM`
`ilogb(x)`	`x == Inf`	`INT_MAX`	`EDOM`
`ilogb(x)`	`x == NaN`	`FP_ILOGBNAN`	`EDOM`
`log(x)`	`x < 0`	`NaN`	`EDOM`
`log(x)`	`x == 0`	`-Inf`	`ERANGE`
`log10(x)`	`x < 0`	`NaN`	`EDOM`
`log10(x)`	`x == 0`	`-Inf`	`ERANGE`
`log1p(x)`	`x < -1`	`NaN`	`EDOM`
`log1p(x)`	`x == -1`	`-Inf`	`ERANGE`
`log2(x)`	`x < 0`	`NaN`	`EDOM`
`log2(x)`	`x == 0`	`-Inf`	`ERANGE`
`logb(x)`	`x == 0`	`-Inf`	`EDOM`
`logb(x)`	`x == Inf`	`+Inf`	`EDOM`
`pow(x, y)`	`y < 0` and (`x == +0` or `y` is even)	`+Inf`	`ERANGE`
`pow(x, y)`	`y < 0` and `x == -0` and `y` is odd	`-Inf`	`ERANGE`
`pow(x, y)`	`y < 0` and `x == -0` and `y` is non-integer	`+Inf`	`ERANGE`
`pow(x,y)`	`x < 0` and `y` is non-integer	`NaN`	`EDOM`
`sqrt(x)`	`x < 0`	`NaN`	`EDOM`
`lgamma(x)`	`x <= 0`	`Inf`	`ERANGE`
`tgamma(x)`	`x < 0` and `x` is integer	`NaN`	`EDOM`
`tgamma(x)`	`x == 0`	`Inf`	`ERANGE`
`fmod(x,y)`	`x == Inf`	`NaN`	`EDOM`
`fmod(x,y)`	`y == 0`	`NaN`	`EDOM`
`remainder(x, y)`	`y == 0`	`NaN`	`EDOM`
`remquo(x, y, q)`	`y == 0`	`NaN`	`EDOM`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The values returned by the mathematics functions on domain errors or pole errors (7.12.1).

See previous table.

The values returned by the mathematics functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling` & `MATH_ERRNO` is nonzero, and whether the Underflow floating-point exception is raised when the integer expression `math_errhandling` & `MATH_ERREXCEPT` is nonzero. (7.12.1).

On underflow, the mathematics functions return `0.0`, the `errno` is set to `ERANGE`, and the Underflow and Inexact exceptions are raised.

The value of `math_errhandling` depends on the mode specified for `-ffp-mode`. For more information, see -ffp-mode.

Whether a domain error occurs or zero is returned when an fmod function has a second argument of zero (7.12.10.1).

When the second argument of `fmod` is zero, a domain error occurs.

Whether a domain error occurs or zero is returned when a remainder function has a second argument of zero (7.12.10.2).

When the second argument of the remainder function is zero, a domain error occurs and the function returns NaN.

The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient (7.12.10.3).

The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient is 4.

Whether a domain error occurs or zero is returned when a `remquo` function has a second argument of zero (7.12.10.3).

When the second argument of the `remquo` function is zero, a domain error occurs.

Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

The equivalent of `signal(sig, SIG_DFL)` is executed before the call to a signal handler.

The null pointer constant to which the macro NULL expands (7.19).

The macro `NULL` expands to 0.

Whether the last line of a text stream requires a terminating new-line character (7.21.2).

The last line of text stream does not require a terminating new-line character.

Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.21.2).

Space characters, written out to a text stream immediately before a new-line character, appear when read back.

The number of null characters that may be appended to data written to a binary stream (7.21.2).

No null characters are appended at the end of a binary stream.

Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.21.3).

The file position indicator of an append-mode stream is positioned initially at the end of the file.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Whether a write on a text stream causes the associated file to be truncated beyond that point (7.21.3).

A write to a text stream causes the associated file to be truncated beyond the point where the write occurred if this is the behavior of the device category of the file.

The characteristics of file buffering (7.21.3).

The C Library supports unbuffered, fully buffered, and line buffered streams.

Whether a zero-length file actually exists (7.21.3).

A zero-length file exists, even if no characters are written by an output stream.

The rules for composing valid file names (7.21.3).

Valid file names depend on the execution environment.

Whether the same file can be simultaneously open multiple times (7.21.3).

A file can be opened many times for reading, but only once for writing or updating.

The nature and choice of encodings used for multibyte characters in files (7.21.3).

The character input and output functions on wide-oriented streams interpret the multibyte characters in the associated files according to the current chosen locale.

The effect of the remove function on an open file (7.21.4.1).

Depends on the environment.

The effect if a file with the new name exists prior to a call to the rename function (7.21.4.2).

Depends on the environment.

Whether an open temporary file is removed upon abnormal program termination (7.21.4.3).

Depends on the environment.

Which changes of mode are permitted (if any), and under what circumstances (7.21.5.4)

No changes of mode are permitted.

The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.21.6.1, 7.29.2.1).

A double argument to the `printf` family of functions, representing an infinity is converted to `[-]inf`. A double argument representing a NaN is converted to `[-]nan`. The F conversion specifier, produces `[-]INF` or `[-]NAN`, respectively.

The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.21.6.1, 7.29.2.1).

The `fprintf` and `fwprintf` functions print %p arguments in lowercase hexadecimal format as if a precision of 8 (16 for 64-bit) had been specified. If the variant form (`%#p`) is used, the number is preceded by the character `@`.


Note

Using the `#` character with the `p` format specifier is undefined behavior in C11. `armclang` issues a warning.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The interpretation of a - character that is neither the first nor the last character, nor the second where a ^ character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.1).

`fscanf` and `fwscanf` always treat the character - in a `%...[...]` argument as a literal character.

The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.2).

`fscanf` and `fwscanf` treat `%p` arguments exactly the same as `%x` arguments.

The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.21.9.1, 7.21.9.3, 7.21.9.4).

On failure, the functions `fgetpos`, `fsetpos`, and `ftell` set the `errno` to `EDOM`.

The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.22.1.3, 7.29.4.1.1).

Any n-char or n-wchar sequence in a string, representing a NaN, that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` functions, is ignored.

Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.22.1.3, 7.29.4.1.1).

The `strtod` , `strtold`, `wcstod`, `wcstof`, or `wcstold` functions set `errno` to `ERANGE` when underflow occurs.

The strtof function sets the errno to `ERANGE` by default (equivalent to compiling with `-ffp-mode=std`) and does not, when compiling with `-ffp-mode=full` or `-fno-fast-math`.

Whether the `calloc`, `malloc`, and `realloc` functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.22.3).

If the size of area requested is zero, `malloc()` and `calloc()` return a pointer to a zero-size block.

If the size of area requested is zero, `realloc()` returns `NULL`.

Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the abort or `_Exit` function is called (7.22.4.1, 7.22.4.5).

The function `_Exit` flushes the streams, closes all open files, and removes the temporary files.

The function `abort()` does not flush the streams and does not remove temporary files.

The termination status returned to the host environment by the `abort`, `exit`, `_Exit()`, or `quick_exit` function (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7).

The function `abort()` returns termination status 1 to the host environment. The functions `exit()` and `_Exit()` return the same value as the argument that was passed to them.

The value returned by the system function when its argument is not a null pointer (7.22.4.8).

The value returned by the system function when its argument is not a null pointer depends on the environment.

The range and precision of times representable in `clock_t` and `time_t` (7.27).

The types `clock_t` and `time_t` can represent integers in the range [0, 4294967295].

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The local time zone and Daylight Saving Time (7.27.1).

Depends on the environment.

The era for the clock function (7.27.2.1).

Depends on the environment.

The `TIME_UTC` epoch (7.27.2.5).

`TIME_UTC` and `timespec_get` are not implemented.

The replacement string for the `%Z` specifier to the `strftime` and `wcsftime` functions in the "C" locale (7.27.3.5, 7.29.5.1).

The functions `strftime` and `wcsftime` replace `%Z` with an empty string.

Whether the functions in `<math.h>` honor the rounding direction mode in an IEC 60559 conformant implementation, unless explicitly specified otherwise (F.10).

Arm Compiler for Embedded does not declare `__STDC_IEC_559__` and does not support Annex F of ISO/IEC 9899:2011.

Related information

The Arm C and C++ Libraries

A.1.15 Architecture

Describes implementation-defined aspects of the Arm C compiler and C library relating to architecture, as required by the ISO C standard.

The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (5.2.4.2, 7.20.2, 7.20.3).


Note

If the value column contains "-", this means no value is assigned to the corresponding macro.

The values of the macros in `<float.h>` are:

Macro name	Value
`FLT_ROUNDS`	1
`FLT_EVAL_METHOD`	0
`FLT_HAS_SUBNORM`	-
`DBL_HAS_SUBNORM`	-
`LDBL_HAS_SUBNORM`	-
`FLT_RADIX`	2
`FLT_MANT_DIG`	24
`DBL_MANT_DIG`	53
`LDBL_MANT_DIG` (AArch32)	53

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Macro name	Value
`LDBL_MANT_DIG` (AArch64)	`113`
`FLT_DECIMAL_DIG`	-
`DBL_DECIMAL_DIG`	-
`LDBL_DECIMAL_DIG` (AArch32)	-
`LDBL_DECIMAL_DIG` (AArch64)	`36`
`DECIMAL_DIG`	`17`
`FLT_DIG`	`6`
`DBL_DIG`	`15`
`LDBL_DIG` (AArch32)	`15`
`LDBL_DIG` (AArch64)	`33`
`FLT_MIN_EXP`	`(-125)`
`DBL_MIN_EXP`	`(-1021)`
`LDBL_MIN_EXP` (AArch32)	`(-1021)`
`LDBL_MIN_EXP` (AArch64)	`16381`
`FLT_MIN_10_EXP`	`(-37)`
`DBL_MIN_10_EXP`	`(-307)`
`LDBL_MIN_10_EXP` (AArch32)	`(-307)`
`LDBL_MIN_10_EXP` (AArch64)	`4931`
`FLT_MAX_EXP`	`128`
`DBL_MAX_EXP`	`1024`
`LDBL_MAX_EXP` (AArch32)	`1024`
`LDBL_MAX_EXP` (AArch64)	`16384`
`FLT_MAX_10_EXP`	`38`
`DBL_MAX_10_EXP`	`308`
`LDBL_MAX_10_EXP` (AArch32)	`308`
`LDBL_MAX_10_EXP` (AArch32)	`4932`
`FLT_MAX`	`3.40282347e+38F`
`DBL_MAX`	`1.79769313486231571e+308`
`LDBL_MAX` (AArch32)	`1.79769313486231571e+308L`
`LDBL_MAX` (AArch64)	`1.189731495357231765085759326628007 02e+4932L`
`FLT_EPSILON`	`1.19209290e-7F`
`DBL_EPSILON`	`2.2204460492503131e-16`
`LDBL_EPSILON` (AArch32)	`2.2204460492503131e-16L`
`LDBL_EPSILON` (AArch64)	`1.92592994438723585305597794258492732e-34L`
`FLT_MIN`	`1.175494351e-38F`
`DBL_MIN`	`2.22507385850720138e-308`
`LDBL_MIN` (AArch32)	`2.22507385850720138e-308L`
`LDBL_MIN` (AArch64)	`3.3621031431120935062626778173217526 0e-4932L`
`FLT_TRUE_MIN`	-
`DBL_TRUE_MIN`	-

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Macro name	Value
LDBL_TRUE_MIN (AArch32)	-
LDBL_TRUE_MIN (AArch64)	16494L

The values of the macros in <limits.h> are:

Macro name	Value (AArch32 state)	Value (AArch64 state)
CHAR_BIT	8	8
SCHAR_MIN	(-128)	(-128)
SCHAR_MAX	127	127
UCHAR_MAX	255	255
CHAR_MIN	0	0
CHAR_MAX	255	255
MB_LEN_MAX	6	6
SHRT_MIN	(-0x8000)	(-0x8000)
SHRT_MAX	0x7fff	0x7fff
USHRT_MAX	65535	65535
INT_MIN	(~0x7fffffff)	(~0x7fffffff)
INT_MAX	0x7fffffff	0x7fffffff
UINT_MAX	0xffffffffU	0xffffffffU
LONG_MIN	(~0x7fffffffL)	(~0x7fffffffffffffffL)
LONG_MAX	0x7fffffffL	0x7fffffffffffffffL
ULONG_MAX	0xffffffffUL	0xffffffffffffffffUL
LLONG_MIN	(~0x7fffffffffffffffLL)	(~0x7fffffffffffffffLL)
LLONG_MAX	0x7fffffffffffffffLL	0x7fffffffffffffffLL
ULLONG_MAX	0xffffffffffffffffULL	0xffffffffffffffffULL

The values of the macros in <stdint.h> are:

Macro name	Value (AArch32 state)	Value (AArch64 state)
INT8_MIN	-128	-128
INT8_MAX	127	127
UINT8_MAX	255	255
INT16_MIN	-32768	-32768
INT16_MAX	32767	32767
UINT16_MAX	65535	65535
INT32_MIN	(~0x7fffffff)	(~0x7fffffff)
INT32_MAX	2147483647	2147483647
UINT32_MAX	4294967295u	4294967295u
INT64_MIN	(~0x7fffffffffffffffLL)	(~0x7fffffffffffffffL)
INT64_MAX	(9223372036854775807LL)	(9223372036854775807L)
UINT64_MAX	(18446744073709551615uLL)	(18446744073709551615uL)
INT_LEAST8_MIN	-128	-128

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Macro name	Value (AArch32 state)	Value (AArch64 state)
INT_LEAST8_MAX	127	127
UINT_LEAST8_MAX	255	255
INT_LEAST16_MIN	-32768	-32768
INT_LEAST16_MAX	32767	32767
UINT_LEAST16_MAX	65535	65535
INT_LEAST32_MIN	(~0x7fffffff)	(~0x7fffffff)
INT_LEAST32_MAX	2147483647	2147483647
UINT_LEAST32_MAX	4294967295u	4294967295u
INT_LEAST64_MIN	(~0x7fffffffffffffffLL)	(~0x7fffffffffffffffL)
INT_LEAST64_MAX	(9223372036854775807LL)	(9223372036854775807L)
UINT_LEAST64_MAX	(18446744073709551615uLL)	(18446744073709551615uL)
INT_FAST8_MIN	(~0x7fffffff)	(~0x7fffffff)
INT_FAST8_MAX	2147483647	2147483647
UINT_FAST8_MAX	4294967295u	4294967295u
INT_FAST16_MIN	(~0x7fffffff)	(~0x7fffffff)
INT_FAST16_MAX	2147483647	2147483647
UINT_FAST16_MAX	4294967295u	4294967295u
INT_FAST32_MIN	(~0x7fffffff)	(~0x7fffffff)
INT_FAST32_MAX	2147483647	2147483647
UINT_FAST32_MAX	4294967295u	4294967295u
INT_FAST64_MIN	(~0x7fffffffffffffffLL)	(~0x7fffffffffffffffL)
INT_FAST64_MAX	(9223372036854775807LL)	(9223372036854775807L)
UINT_FAST64_MAX	(18446744073709551615uLL)	(18446744073709551615uL)
INTPTR_MIN	(~0x7fffffff)	(~0x7fffffffffffffffL)
INTPTR_MAX	2147483647	(9223372036854775807L)
UINTPTR_MAX	4294967295u	(18446744073709551615uL)
INTMAX_MIN	(~0x7fffffffffffffffll)	(~0x7fffffffffffffffll)
INTMAX_MAX	(9223372036854775807ll)	(9223372036854775807ll)
UINTMAX_MAX	(18446744073709551615ull)	(18446744073709551615ull)
PTRDIFF_MIN	(~0x7fffffff)	(~0x7fffffffffffffffL)
PTRDIFF_MAX	2147483647	(9223372036854775807L)
SIG_ATOMIC_MIN	(~0x7fffffff)	(~0x7fffffff)
SIG_ATOMIC_MAX	2147483647	2147483647
SIZE_MAX	4294967295u	(18446744073709551615uL)
WCHAR_MIN	0	0
WCHAR_MAX	0xffffffffU	0xffffffffU
WINT_MIN	(~0x7fffffff)	(~0x7fffffff)
WINT_MAX	2147483647	2147483647

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

The result of attempting to indirectly access an object with automatic or thread storage duration from a thread other than the one with which it is associated (6.2.4).

Access to automatic or thread storage duration objects from a thread other than the one with which the object is associated proceeds normally.

The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).

Defined in the Arm® EABI.

Whether any extended alignments are supported and the contexts in which they are supported, and valid alignment values other than those returned by an `_Alignof` expression for fundamental types, if any (6.2.8).

Alignments, including extended alignments, that are a power of 2 and less than or equal to `0x10000000`, are supported.

The value of the result of the `sizeof` and `_Alignof` operators (6.5.3.4).

Type	`sizeof`	`_Alignof`
`char`	1	1
`short`	2	2
`int`	4	4
`long` (AArch32 state)	4	4
`long` (AArch64 state)	8	8
`long long`	8	8
`float`	4	4
`double`	8	8
`long double` (AArch32 state)	8	8
`long double` (AArch64 state). This data type is not supported for AArch64 state because of limitations in the current Arm C library.	16	16


Note

Arm Compiler for Embedded 6 implements the LP64 data model for AArch64 state.

A.2 Standard C++ Implementation Definition

Provides information required by the ISO C++ Standard for conforming C++ implementations.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A.2.1 Implementation definition (ISO C++ Standard)

The ISO C++ Standard (ISO/IEC 14882:2014) defines the concept of implementation-defined behavior as the "behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents".

The following topics document the behavior in the implementation of Arm® Compiler for Embedded 6 of the implementation-defined features of the C++ language. Each topic provides information from a single chapter in the C++ Standard. The C++ Standard section number relevant to each implementation-defined aspect is provided in parentheses.

A.2.2 General

Describes general implementation-defined aspects of the Arm C++ compiler and C++ library, as required by the ISO C++ Standard.

How a diagnostic is identified (1.3.6).

Diagnostic messages that the compiler produces are of the form:

```
<source-file>:<line-number>:<char-number>: <description> [<diagnostic-flag>]
```

Here:

<description>

Is a text description of the error.

<diagnostic-flag>

Is an optional diagnostic flag of the form `-W<name>`, only for messages that can be suppressed.

Libraries in a freestanding implementation (1.4).

Arm® Compiler for Embedded supports the C99 and the C++11 standard libraries.

Bits in a byte (1.7).

The number of bits in a byte is 8.

What constitutes an interactive device (1.9).

What constitutes an interactive device depends on the environment and what the `_sys_istty` function reports. The standard I/O streams `stdin`, `stdout`, and `stderr` are assumed to be interactive devices. They are line-buffered at program startup, regardless of what `_sys_istty` reports for them. An exception is if they have been redirected on the command line.

Related information

-W (armclang) on page 226

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A.2.3 Lexical conventions

Describes the lexical conventions of implementation-defined aspects of the Arm C++ compiler and C++ library, as required by the ISO C++ Standard.

Mapping of the physical source file characters to the basic source character set (2.2).

The input files are encoded in UTF-8. Due to the design of UTF-8 encoding, the basic source character set is represented in the source file in the same way as the ASCII encoding of the basic character set.

Physical source file characters (2.2).

The source file characters are encoded in UTF-8.

Conversion of characters from source character set to execution character set (2.2).

The source character set and the execution character set are the same.

Requirement of source for translation units when locating template definitions (2.2).

When locating the template definitions related to template instantiations, the source of the translation units that define the template definitions is not required.

Values of execution character sets (2.3).

Both the execution character set and the wide execution character set consist of all the code points defined by ISO/IEC 10646.

Mapping the header name to external source files (2.8).

In both forms of the `#include` preprocessing directive, the character sequences that specify header names are mapped to external header source file names.

Semantics of non-standard escape sequences (2.13.3).

The following non-standard escape sequences are accepted for compatibility with GCC:

Escape sequence	Code point
`\e`	`U+001B`
`\E`	`U+001B`

Value of wide-character literals containing multiple characters (2.13.3).

If a wide-character literal contains more than one character, only the right-most character in the literal is used.

Value of an ordinary character literal outside the range of its corresponding type (2.13.3).

This case is diagnosed and rejected.

Floating literals (2.13.4).

For a floating literal whose scaled value cannot be represented as a floating-point value, the nearest even floating-point value is chosen.

String literal concatenation (2.13.5).

Differently prefixed string literal tokens cannot be concatenated, except for the ones specified by the ISO C++ Standard.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A.2.4 Basic concepts

Describes basic concepts relating to implementation-defined aspects of the Arm C++ compiler and C++ library, as required by the ISO C++ Standard.

Start-up and termination in a freestanding environment (3.6.1).

The Arm® Compiler for Embedded *Arm C and C++ Libraries and Floating-Point Support User Guide* describes the start-up and termination of programs.

Definition of `main` in a freestanding environment (3.6.1).

The `main` function must be defined.

Linkage of the `main` function (3.6.1).

The `main` function has external linkage.

Parameters of `main` (3.6.1).

The only permitted parameters for definitions of `main` of the form `int main(<parameters>)` are `void` and `int, char**`.

Dynamic initialization of static objects (3.6.2).

Static objects are initialized before the first statement of `main`.

Dynamic initialization of thread-local objects (3.6.2).

Thread-local objects are initialized at the first odr-use.

Pointer safety (3.7.4.3).

This implementation has relaxed pointer safety.

Extended signed integer types (3.9.1).

No extended integer types exist in the implementation.

Representation and signedness of the `char` type (3.9.1).

The `char` type is unsigned and has the same values as `unsigned char`.

Representation of the values of floating-point types (3.9.1).

The values of floating-point types are represented using the IEEE format as follows:

- `float` values are represented by IEEE single-precision values.
- `double` values are represented by IEEE double-precision values.
- `long double` values in AArch32 are represented by IEEE double-precision values.
- `long double` values in AArch64 are represented by IEEE quadruple-precision values.


Note

The `long double` data type is not supported for AArch64 state because of limitations in the current Arm C library.

Representation of values of pointer type (3.9.2).

Values of pointer type are represented as 32-bit addresses in AArch32 state and 64-bit addresses in AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Support of extended alignments (3.11).

Alignments, including extended alignments, that are a power of two and are less than or equal to `0x10000000` are supported.

Related information

Arm C and C++ Libraries and Floating-Point Support User Guide

A.2.5 Standard conversions

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to standard conversions, as required by the ISO C++ Standard.

Conversion to signed integer (4.7).

When an integer value is converted to a value of signed integer type, but cannot be represented by the destination type, the value is truncated to the number of bits of the destination type and then reinterpreted as a value of the destination type.

Result of inexact floating-point conversions (4.8).

When a floating-point value is converted to a value of a different floating-point type, and the value is within the range of the destination type but cannot be represented exactly, the value is rounded to the nearest floating-point value by default.

Result of inexact integer to floating-point conversion (4.9).

When an integer value is converted to a value of floating-point type, and the value is within the range of the destination type but cannot be represented exactly, the value is rounded to the nearest floating-point value by default.

A.2.6 Expressions

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to expressions, as required by the ISO C++ Standard.

Passing an argument of class type in a function call through ellipsis (5.2.2).

For ellipsis arguments, passing an argument of class type having a non-trivial copy constructor, a non-trivial move constructor, or a non-trivial destructor, with no corresponding parameter, results in an abort at run time. A diagnostic is reported for this case.

Result type of `typeid` expression (5.2.8).

The type of a `typeid` expression is an expression with dynamic type `std::type_info`.

Incrementing a bit-field that cannot represent the incremented value (5.2.6).

The incremented value is truncated to the number of bits in the bit-field. The bit-field is updated with the bits of the truncated value.

Conversions between pointers and integers (5.2.10).

Converting a pointer to an integer type with a smaller bit width than the pointer, truncates the pointer to the number of bits of the destination type. Converting a pointer to an integer

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

type with a greater bit width than the pointer, zero-extends the pointer. Otherwise, the bits of the representation are unchanged.

Converting an unsigned integer to a pointer type with a greater bit-width than the unsigned integer zero-extends the integer. Converting a signed integer to a pointer type with a greater bit-width than the signed integer sign-extends the integer. Otherwise, the bits of the representation are unchanged.

Conversions from function pointers to object pointers (5.2.10).

Such conversions are supported.

`sizeof` applied to fundamental types other than `char`, `signed char`, and `unsigned char` (5.3.3).

Type	`sizeof`
`bool`	1
`char`	1
`wchar_t`	4
`char16_t`	2
`char32_t`	4
`short`	2
`int`	4
`long` (AArch32 state)	4
`long` (AArch64 state)	8
`long long`	8
`float`	4
`double`	8
`long double` (AArch32 state)	8
`long double` (AArch64 state). This data type is not supported for AArch64 state because of limitations in the current Arm C library.	16


Note

Arm® Compiler for Embedded 6 implements the LP64 data model for AArch64 state.

Support for over-aligned types in `new` expressions (5.3.4).

Over-aligned types are not supported in `new` expressions. The pointer for the allocated type will not fulfill the extended alignment.

Type of `ptrdiff_t` (5.7).

The type of `ptrdiff_t` is `signed int` for AArch32 state and `signed long` for AArch64 state.

Type of `size_t` (5.7).

The type of `size_t` is `unsigned int` for AArch32 state and `unsigned long` for AArch64 state.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Result of right shift of negative value (5.8).

In a bitwise right shift operation of the form `E1 >> E2`, if `E1` is of signed type and has a negative value, the value of the result is the integral part of the quotient of `E1 / (2 ** E2)`, except when `E1` is -1, then the result is -1.

Assignment of a value to a bit-field that the bit-field cannot represent (5.18).

When assigning a value to a bit-field that the bit-field cannot represent, the value is truncated to the number of bits of the bit-field. A diagnostic is reported in some cases.

Related information

Basic concepts on page 1067

A.2.7 Declarations

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to declarations, as required by the ISO C++ Standard.

Meaning of attribute declaration (7).

Arm® Compiler for Embedded 6 is based on LLVM and Clang technology. Clang defines several attributes as specified by the Clang documentation at https://clang.llvm.org/docs/AttributeReference.html.

From these attributes, Arm Compiler for Embedded 6 supports attributes that are scoped with `gnu::` (for compatibility with GCC) and `clang::`.

Underlying type for enumeration (7.2).

The underlying type for enumerations without a fixed underlying type is `int` or `unsigned int`, depending on the values of the enumerators. The `-fshort-enums` command-line option uses the smallest unsigned integer possible, or the smallest signed integer possible if any enumerator is negative, starting with `char`.

Meaning of an `asm` declaration (7.4).

An `asm` declaration enables the direct use of T32, A32, or A64 instructions.

Semantics of linkage specifiers (7.5).

Only the string-literals "C" and "C++" can be used in a linkage specifier.

A.2.8 Declarators

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to declarators, as required by the ISO C++ Standard.

String resulting from `__func__` (8.4.1).

The value of `__func__` is the same as in C99.

Initialization of a bit-field with a value that the bit-field cannot represent (8.5).

When initializing a bit-field with a value that the bit-field cannot represent, the value is truncated to the number of bits of the bit-field. A diagnostic is reported in some cases.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Allocation of bit-fields within a class (9.6).

Within a storage unit, successive bit-fields are allocated from low-order bits towards high-order bits when compiling for little-endian, or from the high-order bits towards low-order bits when compiling for big-endian.

Alignment of bit-fields within a class (9.6).

The storage unit containing the bit-fields is aligned to the alignment of the type of the bit-field.

A.2.9 Templates

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to templates, as required by the ISO C++ Standard.

Linkage specification in templates (14).

Only the linkage specifiers "C" and "C++" can be used in template declarations.

A.2.10 Exception handling

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to exception handling, as required by the ISO C++ Standard.

Stack unwinding before calling `std::terminate` when no suitable catch handler is found (15.3).

The stack is not unwound in this case.

Stack unwinding before calling `std::terminate` when a noexcept specification is violated (15.5.1).

The stack is unwound in this case.

A.2.11 Preprocessing directives (ISO C++ Standard)

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to preprocessing directives, as required by the ISO C++ Standard.

Numeric values of character literals in `#if` preprocessing directives (16.1).

Numeric values of character literals match the values that they have in expressions other than the `#if` or `#elif` preprocessing directives.

Sign of character literals in `#if` preprocessing directives (16.1).

Character literals in `#if` preprocessing directives are never negative.

Manner in which #include <. . .> source files are searched (16.2).

- If the character sequence begins with the `/` character, it is interpreted as an absolute file path.
- Otherwise, the character sequence is interpreted as a file path relative to one of the following directories:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The sequence of the directories specified using the `-I` command-line option, in the command-line order.
- The `include` subdirectory in the compiler installation directory.

Manner in which `#include "..."` source files are searched (16.2).

- If the character sequence begins with the `/` character, it is interpreted as an absolute file path.
- Otherwise, the character sequence is interpreted as a file path relative to the parent directory of the source file that contains the `#include` preprocessing directive.

Nesting limit for `#include` preprocessing directives (16.2).

Limited only by the memory available at translation time.

Meaning of pragmas (16.6).

Arm® Compiler for Embedded 6 is based on LLVM and Clang technology. Clang defines several pragmas as specified by the Clang documentation at http://clang.llvm.org/docs/LanguageExtensions.html.

Definition and meaning of `__STDC__` (16.8).

`__STDC__` is predefined as `#define __STDC__ 1`.

Definition and meaning of `__STDC_VERSION__` (16.8).

This macro is not predefined.

Text of `__DATE__` and `__TIME__` when the date or time of a translation is not available (16.8).

The date and time of the translation are always available on all supported platforms.

A.2.12 Library introduction

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the library introduction, as required by the ISO C++ Standard.

Linkage of names from the Standard C library (17.6.2.3).

Declarations from the C library have "C" linkage.

Library functions that can be recursively reentered (17.6.5.8).

Functions can be recursively reentered, unless specified otherwise by the ISO C++ Standard.

Exceptions thrown by C++ Standard Library functions that do not have an exception specification (17.6.5.12).

These functions do not throw any additional exceptions.

Errors category for errors originating from outside the operating system (17.6.5.14).

There is no additional error category.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A.2.13 Language support library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the language support library, as required by the ISO C++ Standard.

Exit status (18.5).

Control is returned to the host environment using the `_sys_exit` function of the Arm C Library.

Returned value of `std::bad_alloc::what` (18.6.2.1).

The returned value is `std::bad_alloc`.

Returned value of `std::type_info::name` (18.7.1).

The returned value is a string containing the mangled name of the type that is used in the `typeid` expression. The name is mangled following the Itanium C++ ABI specification.

Returned value of `std::bad_cast::what` (18.7.2).

The returned value is `std::bad_cast`.

Returned value of `std::bad_typeid::what` (18.7.3).

The returned value is `std::bad_typeid`.

Returned value of `std::bad_exception::what` (18.8.1).

The returned value is `std::bad_exception`.

Returned value of `std::exception::what` (18.8.1).

The returned value is `std::exception`.

Use of non-POFs as signal handlers (18.10).

Non Plain Old Functions (POFs) can be used as signal handlers if no uncaught exceptions are thrown in the handler, and the execution of the signal handler does not trigger undefined behavior. For example, the signal handler might have to call `std::_Exit` instead of `std::exit`.

A.2.14 General utilities library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the general utilities library, as required by the ISO C++ Standard.

Return value of `std::get_pointer_safety` (20.7.4).

This function always returns `std::pointer_safety::relaxed`.

Support for over-aligned types by the allocator (20.7.9.1).

The allocator does not support over-aligned types.

Support for over-aligned types by `get_temporary_buffer` (20.7.11).

Function `std::get_temporary_buffer` does not support over-aligned types.

Returned value of `std::bad_weak_ptr::what` (20.8.2.2.1).

The returned value is `bad_weak_ptr`.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Exception type when the constructor of `std::shared_ptr` fails (20.8.2.2.1).

`std::bad_alloc` is the only exception that the `std::shared_ptr` constructor throws that receives a pointer.

Placeholder types (20.9.10.4).

Placeholder types, such as `std::placeholders::_1`, are not `CopyAssignable`.

Over-aligned types and type traits `std::aligned_storage` and `std::aligned_union` (20.10.7.6).

These two traits support over-aligned types.

Conversion between `time_t` and `time_point` (20.12.7.1).

The values are truncated in either case.

A.2.15 Strings library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the strings library, as required by the ISO C++ Standard.

Type of `std::streamoff` (21.2.3.1).

Type `std::streamoff` has type `long long`.

Type of `std::streampos` (21.2.3.2).

Type of `std::streampos` is `fpos<mbstate_t>`.

Returned value of `char_traits<char16_t>::eof` (21.2.3.2).

This function returns `uint_least16_t(0xFFFF)`.

Type of `std::u16streampos` (21.2.3.3).

Type of `std::u16streampos` is `fpos<mbstate_t>`.

Returned value of `char_traits<char32_t>::eof` (21.2.3.3).

This function returns `uint_least32_t(0xFFFFFFFF)`.

Type of `std::u32streampos` (21.2.3.3).

Type of `std::u32streampos` is `fpos<mbstate_t>`.

Type of `std::wstreampos` (21.2.3.4).

Type of `std::wstreampos` is `fpos<mbstate_t>`.

Supported multibyte character encoding rules (21.2.3.4).

UTF-8 and Shift-JIS are supported as multibyte character encodings.

A.2.16 Numerics library

The Arm implementation of the C++ standard library class `std::random_device` is different from the open-source libc++ implementation.

The Arm C++ standard library implementation has the following features:

- For the constructor `std::random_device`, implementation limitations prevent generating nondeterministic random numbers. The implementation uses a pseudo-random number engine.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

- The default constructor of `std::random_device` uses the token `"pseudorandom"`.
- The only valid value of the parameter token of the `std::random_device` constructor `explicit random_device(const string &token)` is `"pseudorandom"`.
- If there is a failure, the type of exceptions that `std::random_device` constructors throw is `std::system_error`.
- The member function `result_type operator()()` of `std::random_device` never fails.

A.2.17 Localization library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the localization library, as required by the ISO C++ Standard.

Locale object (22.3.1.2).

There is one global locale object for the entire program.

Permitted locale names (22.3.1.2).

Valid locale values depend on which `__use_X_ctype` symbols are imported (`__use_iso8859_ctype`, `__use_sjis_ctype`, `__use_utf8_ctypte`), and on user-defined locales.

Effect on C locale of `calling locale::global` (22.3.1.5).

Calling this function with an unnamed locale has no effect.

Value of `ctype<char>::table_size` (22.4.1.3.1).

The value of `ctype<char>::table_size` is 256.

Two-digit year numbers in the function `std::time_get::do_get_year` (22.4.5.1.2).

Two-digit year numbers are accepted. Years from 00 to 68 are assumed to mean years 2000 to 2068, while years from 69 to 99 are assumed to mean 1969 to 1999.

Additional formats for `std::time_get::do_get_date` (22.4.5.1.2).

No additional formats are defined.

Formatted character sequence that `std::time_put::do_put` generates in the C locale (22.4.5.3.2).

The behavior is the same as that of the Arm C library function `strftime`.

Mapping from name to catalog when calling `std::messages::do_open` (22.4.7.1.2).

No mapping happens as this function does not open any catalog.

Mapping to message when calling `std::messages::do_get` (22.4.7.1.2).

No mapping happens and `dflt` is always returned.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A.2.18 Containers library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the containers library, as required by the ISO C++ Standard.

Type of `std::array::iterator` and `std::array::const_iterator` (23.3.2.1).

The types of `std::array<T>::iterator` and `std::array<T>::const_iterator` are `T*` and `const T*` respectively.

Default number of buckets in `std::unordered_map` (23.5.4.2).

When constructing a container with an iterator range and without specifying the number of buckets, the number of buckets that are used is equal to the size of the iterator range. Every element of the iterator range is inserted in an empty container.

Default number of buckets in `std::unordered_multimap` (23.5.4.2).

When constructing a container with an iterator range and without specifying the number of buckets, the number of buckets that are used is equal to the size of the iterator range. Every element of the iterator range is inserted in an empty container.

Default number of buckets in `std::unordered_set` (23.5.6.2).

When constructing a container with an iterator range and without specifying the number of buckets, the number of buckets that are used is equal to the size of the iterator range. Every element of the iterator range is inserted in an empty container.

Default number of buckets in `std::unordered_multiset` (23.5.7.2).

When constructing a container with an iterator range and without specifying the number of buckets, the number of buckets that are used is equal to the size of the iterator range. Every element of the iterator range is inserted in an empty container.

A.2.19 Input/output library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the input/output library, as required by the ISO C++ Standard.

Behavior of `iostream` classes when `traits::pos_type` is not `streampos` or when `traits::off_type` is not `streamoff` (27.2.1).

There is no specific behavior implemented for this case.

Effect of calling `std::ios_base::sync_with_stdio` after any input or output operation on standard streams (27.5.3.4).

Previous input/output is not handled in any special way.

Exception thrown by `basic_ios::clear` (27.5.5.4).

When `basic_ios::clear` throws as exception, it throws an exception of type `basic_ios::failure` constructed with `"ios_base::clear"`.

Move constructor of `std::basic_stringbuf` (27.8.2.1).

The constructor copies the sequence pointers.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Effect of calling `std::basic_filebuf::setbuf` with nonzero arguments (27.9.1.2).

The provided buffer replaces the internal buffer. The object can use up to the provided number of bytes of the buffer.

Effect of calling `std::basic_filebuf::sync` when a get area exists (27.9.1.5).

The get area is emptied and the current file position is moved back the corresponding number of bytes.

`basic_stringbuf` default constructor `stringbuf.cons` uses the *Small String Optimization* (SSO) (31.8.2.2).

The `basic_stringbuf` sequence pointers are not initialized to `nullptr`:

- `eback()`
- `gptr()`
- `egptr()`
- `pbase()`
- `pptr()`
- `epptr()`

A.2.20 Regular expressions library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the regular expressions library, as required by the ISO C++ Standard.

Type of `std::regex_constants::error_type`

The enum `std::regex_constants::error_type` is defined as follows:

```
enum error_type
{
    error_collate = 1,
    error_ctype,
    error_escape,
    error_backref,
    error_brack,
    error_paren,
    error_brace,
    error_badbrace,
    error_range,
    error_space,
    error_badrepeat,
    error_complexity,
    error_stack,
    __re_err_grammar,
    __re_err_empty,
    __re_err_unknown
};
```

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

A.2.21 Atomic operations library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the atomic operations library, as required by the ISO C++ Standard. To use these macros, specify `#include <atomic>`.

Values of `ATOMIC_...LOCK_FREE` macros (29.4)

Macro	Value
ATOMIC_BOOL_LOCK_FREE	2
ATOMIC_CHAR_LOCK_FREE	2
ATOMIC_CHAR16_T_LOCK_FREE	2
ATOMIC_CHAR32_T_LOCK_FREE	2
ATOMIC_WCHAR_T_LOCK_FREE	2
ATOMIC_SHORT_LOCK_FREE	2
ATOMIC_INT_LOCK_FREE	2
ATOMIC_LONG_LOCK_FREE	2
ATOMIC_LLONG_LOCK_FREE	2
ATOMIC_POINTER_LOCK_FREE	2

A.2.22 Thread support library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the thread support library, as required by the ISO C++ Standard.

Presence and meaning of `native_handle_type` and `native_handle`.

The library uses the following native handles as part of the thread portability mechanism, which is described elsewhere.

```
__ARM_TPL_mutex_t used in std::mutex and std::recursive_mutex
__ARM_TPL_condvar_t used in std::condition_variable
__ARM_TPL_thread_id used in std::thread
__ARM_TPL_thread_t used in std::thread
```

A.2.23 Implementation quantities

Describes limits in C++ implementations.



Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential


Note

Where a specific number is provided, this value is the recommended minimum quantity.

Nesting levels of compound statements, iteration control structures, and selection control structures.

256. Can be increased using the `-fbracket-depth` command-line option.

Nesting levels of conditional inclusion

Limited by memory.

Pointer, array, and function declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration.

Limited by memory.

Nesting levels of parenthesized expressions within a full-expression.

256. Can be increased using the `-fbracket-depth` command-line option.

Number of characters in an internal identifier or macro name.

Limited by memory.

Number of characters in an external identifier.

Limited by memory.

External identifiers in one translation unit.

Limited by memory.

Identifiers with block scope declared in one block.

Limited by memory.

Macro identifiers that are simultaneously defined in one translation unit.

Limited by memory.

Parameters in one function definition.

Limited by memory.

Arguments in one function call.

Limited by memory.

Parameters in one macro definition.

Limited by memory.

Arguments in one macro invocation.

Limited by memory.

Characters in one logical source line.

Limited by memory.

Characters in a string literal (after concatenation).

Limited by memory.

Size of an object.

`SIZE_MAX`

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Nesting levels for `#include` files.

Limited by memory.

Case labels for a switch statement (excluding case labels for any nested switch statements).

Limited by memory.

Data members in a single class.

Limited by memory.

Enumeration constants in a single enumeration.

Limited by memory.

Levels of nested class definitions in a single member-specification.

256. Can be increased using the `-fbracket-depth` command-line option.

Functions that are registered by `atexit()`.

Limited by memory.

Direct and indirect base classes.

Limited by memory.

Direct base classes for a single class.

Limited by memory.

Members declared in a single class.

Limited by memory.

Final overriding virtual functions in a class, accessible or not.

Limited by memory.

Direct and indirect virtual bases of a class.

Limited by memory.

Static members of a class.

Limited by memory.

Friend declarations in a class.

Limited by memory.

Access control declarations in a class.

Limited by memory.

Member initializers in a constructor definition.

Limited by memory.

Scope qualifications of one identifier.

Limited by memory.

Nested external specifications.

Limited by memory.

Recursive `constexpr` function invocations.

512. Can be changed using the [COMMUNITY] command-line option, `-fconstexpr-depth`.

Full-expressions that are evaluated within a core constant expression.

Limited by memory.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Template arguments in a template declaration.

Limited by memory.

Recursively nested template instantiations, including substitution during template argument deduction (14.8.2).

1024. Can be changed using the [COMMUNITY] command-line option, `-ftemplate-depth`.

Handlers per `try` block.

Limited by memory.

Throw specifications on a single function declaration.

Limited by memory.

Number of placeholders (20.9.10.4).

Ten placeholders from `_1` to `_10`.

A.3 Via File Syntax

Via files are plain text files that allow you to specify command-line arguments and options for the `armasm`, `armlink`, `fromelf`, and `armar` tools.

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`. Therefore, you can call multiple nested via files from within a via file.

Via file evaluation

When you invoke the `armasm`, `armlink`, `fromelf`, or `armar`, the tool:

1. Replaces the first specified `--via <via_file>` argument with the sequence of argument words that are 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 that you specify them. Each via file is processed completely, including any nested via files contained in that file, before processing the next via file.

Related information

Via file syntax rules on page 1083
--via=filename (armasm) on page 896

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

--via=filename (armlink) on page 558
--via=file (fromelf) on page 831
--via=filename (armar) on page 854

A.3.1 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:

  `--bigend --debug` (two words)

  `--bigend--debug` (one word)
- The end of a line is treated as whitespace, for example:

  ```
  --bigend
  --debug
  ```

  This is equivalent to:

  `--bigend --debug`
- 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:

  `--errors C:\My Project\errors.txt` (three words)

  `--errors "C:\My Project\errors.txt"` (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:

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

```
--errors "C:\Project\errors.txt"
```

This is treated as the single word:

```
--errorsC:\Project\errors.txt
```

- 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.

Related information

--via=filename (armasm) on page 896
--via=filename (armlink) on page 558
--via=file (fromelf) on page 831
--via=filename (armar) on page 854

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Proprietary Notice

This document is protected by copyright and other related rights and the use or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm Limited ("Arm"). No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm’s trademark usage guidelines at https://www.arm.com/company/policies/trademarks. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

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

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
0623-01	16 October 2024	Non-Confidential	Arm Compiler for Embedded v6.23 Release.
0622-00	13 March 2024	Non-Confidential	Arm Compiler for Embedded v6.22 Release.
0621-00	11 October 2023	Non-Confidential	Arm Compiler for Embedded v6.21 Release.
0620-00	15 March 2023	Non-Confidential	Arm Compiler for Embedded v6.20 Release.
0619-00	12 October 2022	Non-Confidential	Arm Compiler for Embedded v6.19 Release.
0618-01	12 August 2022	Non-Confidential	Documentation update 1 for Arm Compiler for Embedded v6.18 Release.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Issue	Date	Confidentiality	Change
0618-00	22 March 2022	Non-Confidential	Arm Compiler for Embedded v6.18 Release.
0617-00	20 October 2021	Non-Confidential	Arm Compiler for Embedded v6.17 Release.
0616-01	12 March 2021	Non-Confidential	Documentation update 1 for Arm Compiler v6.16 Release.
0616-00	3 March 2021	Non-Confidential	Arm Compiler v6.16 Release.
0615-01	14 December 2020	Non-Confidential	Documentation update 1 for Arm Compiler v6.15 Release.
0615-00	7 October 2020	Non-Confidential	Arm Compiler v6.15 Release.
0614-00	26 February 2020	Non-Confidential	Arm Compiler v6.14 Release.
0613-00	9 October 2019	Non-Confidential	Arm Compiler v6.13 Release.

Change history

The first table is for the first release. Then, each table compares the new issue of the manual with the last released issue of the manual. Release numbers match the revision history in Document release information on page 1087.

Table 2: Issue 0615-01

Change	Topics affected
Removed the [BETA] indications for *Custom Datapath Extension* (CDE), which is fully supported.	• -march. • -mcpu. • --coprocN=value (fromelf).
Added topic for `-dangling-debug-address`.	• --dangling-debug-address=address.
Add the `-mfpu=softvfp` option to the `-mfpu` reference page. This option had been omitted from the documentation previously in error.	• -mfpu.
Added note about not specifying both the architecture (`-march`) and the processor (`-mcpu`).	• -march. • -mcpu.
Added SVE information.	• -ffp-mode. • --target.
Mentioned that dynamic linking is not supported for Cortex®-M processors.	• -fsysv, -fno-sysv. • --sysv.
Added note that an error is now reported if sections of different types share the same name.	• Section directives.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Corrections to the instructions available when `-mcmse` used, and clarification of meanings for the `__ARM_FEATURE_CMSE` values.	• -mcmse. • Predefined macros.
Added note and examples explaining the effect of using different combinations of the `armclang` command-line options `-fno-builtin` and `-nostdlib` when using the standard C and C++ libraries.	• -fno-builtin.
Added note that Arm Compiler for Embedded 6 implements the LP64 data model for AArch64 state to tables showing the result of the `sizeof` and `_Alignof` operators for different types.	• Architecture. • Expressions.
Added note that an error is produced if non-unique section names are specified using the section attribute.	• __attribute__((section("name"))) function attribute. • __attribute__((section("name"))) variable attribute.
Added notes that `vfpv3xd`, `fpv4-sp-d16`, and `fpv5-sp-d16` are single-precision only to the `-mfpu` reference page.	• -mfpu.
Added notes that the `armclang` options `-M`, `-MM`, `-MD`, and `-MMD` do not include files added using the `INCBIN`, `INCLUDE`, or `GET` directives with `armasm`, or the GNU assembly `.incbin` or `.include` directives with the `armclang` integrated assembler.	• -M, -MM. • -MD, -MMD.
Added note that an error is produced if non-unique section names are specified using `#pragma clang section`.	• #pragma clang section.
Improved explanation of when to use the volatile keyword to prevent unwanted removal of inline assembler code when building optimized output.	• Inline Assembly.
Added details of the new `armclang` option `-Omin` to the `-O` topic, added a description of the `armlink` option `-Omin`, added the `min` option to the description of the `armlink` option `--lto-level`.	• -O (armclang). • -Omin (armlink). • --lto_level.
Added Armv8.7-A information. This architecture has [ALPHA]-level support in this release.	• -march. • -mcpu.
Updated Armv8-R information to add 64-bit [BETA]-level support.	• -march.
Added details of the s, j, x, N, and O constraints.	• Constraint codes common to AArch32 state and AArch64 state. • Constraint codes for AArch32 state. • Constraint codes for AArch64 state.
Improved description of the `armlink` option `--sort=AlignmentLexical`.	• --sort=algorithm.
Added a note for the `armlink` option `--sort` not including data sections.	• --sort=algorithm.
Document `nomerge` and `not_tail_called` attributes as [COMMUNITY] features.	• Compiler-specific Function, Variable, and Type Attributes. • __attribute__((nomerge)) function attribute. • __attribute__((not_tail_called)) function attribute.
Improved description of `armclang` command-line option `-save-temps`.	• -save-temps.
Fixes / clarifications to description of `-ffunction-sections` command-line option.	• -ffunction-sections, -fno-function-sections.
Clarify that `armlink` does not `OVERALIGN` some sections where it might be unsafe to do so.	• Alignment of execution regions and input sections. • Syntax of an input section description.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Progressive terminology commitment added to Proprietary notices section (all documents).	• Proprietary notices.

Table 3: Issue 0616-00

Change	Topics affected
Added new Armv8.7-A architecture extensions. Armv8.7-A and the `ls64` feature are now fully supported.	• -march. • -mcpu.
Improved the description for *Prevention of Speculative execution and data prediction*.	• -mcpu.
Added information about linking objects compiled with different C or C++ standards.	• -std.
A note has been added to include a `.balign` directive when defining your own sections with the armclang integrated assembler.	• armclang Integrated Assembler. • Syntax of assembly files for integrated assembler. • Alignment directives.
Updated the description of `-marm` command-line option to clarify that it gives an error, not a warning, when used with an M-profile architecture.	• -marm.
Corrected the LR_2 load region description.	• SysV standard memory model.
Updated the description of `ffp-mode` to explain how the `__use_accurate_btod` and `__use_embedded_btod` symbols affect runtime binary to decimal conversion.	• -ffp-mode.
Updated to clarify the handling of NaNs and infinities.	• -ffp-mode.
Updated warning note about section naming to clarify that section names must not conflict with other program symbols such as function or variable names.	• __attribute__((section("name"))) function attribute. • __attribute__((section("name"))) variable attribute. • #pragma clang section.
Improved description of `-nostdlib` armclang command-line option.	• -nostdlib.
Updated code example in the section of the Inline Assembly chapter which discusses duplication of labels in inline assembly. The new example is simpler and does not rely on C++ 11.	• Duplication of labels in inline assembly statements.
Added support for enabling the pointer authentication Extension (`+pauth`) and Flag Manipulation Instructions (`+flagm`) to the `-march` and `-mcpu` command-line options.	• -march. • -mcpu.
Added a note for the workaround when entry functions or Non-secure function calls have more than 4 arguments.	• -mcmse.
Enhancements to show the relationship between the default armclang-generated sections and scatter-loading input sections.	• Relationship between the default armclang-generated sections and scatter-loading input sections. • Syntax of an input section description.
Clarify that char and short along with other integer types are allowable as bitfield types.	• Structures, unions, enumerations, and bitfields.
Updated to reflect the change in behavior of literal pool merging when using the `PROTECTED` load region attribute.	• --merge, --no_merge. • --merge_litpools, --no_merge_litpools. • Merging identical constants. • Load region attributes. • Execution region attributes.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Table 4: Issue 0617-00

Change	Topics affected
Added a note that `-mthumb` can reduce code size where supported.	• -marm. • -mthumb.
Improved `#pragma clang section` documentation.	• #pragma clang section.
Added a note about `.n` suffix being ignored for some instructions.	• armclang inline assembler. • armclang Integrated Assembler.
Added information about the Armv8.1-M PACBTI extension.	• -mbranch-protection. • -mcpu. • __attribute__((target("options"))) function attribute. • Predefined macros. • --library_security=protection.
Added documentation for `--[no_]summary-stderr`.	• --errors=filename. • --info=topic[,topic,...] (armlink). • --summary_stderr, --no_summary_stderr.
Updated the descriptions of `-march` and `-mcpu` to add details of SME, *Realm Management Extension* (RME), Armv8.8-A, and Armv9.x-A support.	• -march. • -mcpu.
Updated the documentation of *Link-Time Optimization* (LTO). Bit-code libraries can now be used, but only if all libraries are compiled using the same version of the compiler.	• -flto, -fno-lto. • -O (armclang). • --lto, --no_lto.
Added a note about unsupported features not working across tool boundaries.	• -g, -g<level>, -gdwarf-<version> (armclang).
Updated the description of `-mcmse` to include note about not mixing objects with CMSE.	• -mcmse.
Updated *Standard C Implementation Definition* to state that if a wide-character constant contains more than one multibyte character, the compiler now reports an error.	• Characters.
Added information for `-frtti`, `-fno-rtti`.	• -frtti, -fno-rtti.
Added information for *thread local storage* (TLS) support.	• -ftls-model. • -mtls-size. • -mtp. • __attribute__((tls_model("model"))) variable attribute. • *Bare-metal and DLL-like memory models*. • Thread local storage in the bare metal and shared library linking models. • Thread local storage in the SysV linking model.
Added a note that, in a Linux environment, `armlink` requires quotation marks around options that accept parentheses as values.	• Arm Compiler for Embedded tool command-line syntax. • --entry=location. • --first=section_id. • --keep=section_id (armlink). • --last=section_id. • --xreffrom, --xrefto.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Updated description of `-moutline`, `-mno-outline` to include AArch32 support, and updated example.	• -moutline, -mno-outline.
Removed note about `-Omin` not being standards compliant.	• -O (armclang). • --lto_level.
Added notes about build attribute compatibility checking being supported only for AArch32.	• ATTR directive. • --force_explicit_attr.
Bare-metal *Position Independent Executable* (PIE) is no longer deprecated and is supported for both AArch64 state and AArch32 state.	• armclang Command-line Options. • -fbare-metal-pie. • -fropi, -fno-ropi. • -frwpi, -fno-rwpi. • --bare_metal_pie. • --fpic. • --lto_relocation_model. • --pie. • --piveneer, --no_piveneer. • --ropi. • --rwpi. • Generation of position independent to absolute veneers. • Overview of linking models. • Load region attributes. • Execution region attributes.
Added a note that `armclang` always applies the rules for type auto-deduction from C++17, regardless of which C++ source language mode a program is compiled for.	• -std
Added a restrictions section that for AArch64 the `.arch` and `.cpu` assembler directives that do not explicitly include or exclude an extension might have no effect.	• AArch64 target selection directives.
Updated the SysV scatter file support for `ARM_LIB_STACKHEAP`, `ARM_LIB_STACK`, or `ARM_LIB_HEAP` keywords.	• --bare_metal_sysv. • --sysv. • SysV linking model. • Linker command-line options for the SysV linking model. • Requirements and restrictions for using scatter files with SysV linking model. • SysV linking model overview. • Placing the stack and heap with a scatter file.
Corrected the IEEE compliance statements for `fz` libraries.	• -ffast-math, -fno-fast-math. • -ffp-mode.
Updated the description of translation limits. There is no longer a 2GB limit. Arm Compiler for Embedded 6 imposes no limit, other than the limit imposed by available memory.	• Translation limits.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Removed `--reduce_paths` / `--no_reduce_paths` reference information. The `armasm` command-line option `--reduce_paths` provided a way to reduce the length of relative pathnames whose absolute pathnames expand to longer than 260 characters on Windows. Because this only works on 32-bit Windows, Arm recommends instead that you avoid using long and deeply nested file paths.	• armasm Legacy Assembler Reference
Corrected the information in the *Half-precision floating-point data types* section.	• Half-precision floating-point data types.
Added the description of the `-fsized-deallocation`, `-fno-sized-deallocation` command-line options.	• -fsized-deallocation, -fno-sized-deallocation. • armclang Command-line Options.
Clarified the complex number support.	• Support level definitions.
Added a description for the `armclang` option `-d`.	• -d (armclang).
Added note that `-mcmodel` options are only supported on AArch64 targets.	• -mcmodel.
Added note about `armlink` reporting undefined symbol even if unused section removal has removed the requirement for this symbol.	• Elimination of unused sections.
Corrected the descriptions of `long double` IEEE precision for AArch64.	• Architecture. • Floating-point. • Basic concepts. • Expressions.
Added a note for the `armclang` option `-Oz` relating to literal pools.	• -O (armclang).
Added note about using attribute **UNDEFINED** for an undefined instruction handler.	• __attribute__((interrupt("type"))) function attribute.

Table 5: Issue 0618-00

Change	Topics affected
Updated description of the `pauth` architectural feature option to clarify its use with branch protection.	• -march. • -mbranch-protection. • -mcpu.
Clarify rounding method descriptions.	• Floating-point. • Floating-point data definition directives.
Added table showing useful optimization levels for different optimization goals (smaller code size versus improved performance).	• -O (armclang).
Added information for the minimal set of `.cfi` directives that are required for debugging code that does not make use of the stack. Also, modified the description of *-g*, *-gdwarf-2*, *-gdwarf-3*, *-gdwarf-4* to mention using these directives for debugging GNU-syntax assembly source code.	• Call Frame Information directives. • -g, -g<level>, -gdwarf-<version> (armclang).
Added support for the Performance Monitor Extension v3 (PMUv3) for Armv8-A targets, `+pmuv3` and `+nopmuv3`, to the `-march` and `-mcpu` command-line options.	• -march. • -mcpu.
Added support for the A-profile Hinted Conditional Branches Extension, `+hbc` and `+nohbc`, to the `-march` and `-mcpu` command-line options.	• -march. • -mcpu.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Added support for the A-profile Memory Operations Extension, `+mops` and `+nomops`, to the `-march` and `-mcpu` command-line options, and predefined macro `__ARM_FEATURE_MOPS`.	• -march. • -mcpu. • Predefined macros.
Updated descriptions of the `+pactbi` architectural feature option because it is now fully supported.	• -march. • -mcpu.
Added description of the new `armlink` option `--require-bti`, the new `--info=bti` option, and the change in behavior when linking BTI with non-BTI user objects.	• --info=topic[,topic,...] (armlink). • --library_security=protection. • --require-bti.
Added description of the new `armclang` option `-ftrivial-auto-var-init` and the `__attribute((uninitialized))` variable attribute.	• -ftrivial-auto-var-init. • __attribute__((uninitialized)) variable attribute.
Added note that the inline and integrated assemblers incorrectly fail to report an error for a T32 instruction with an invalid `.n` width specifier.	• armclang inline assembler. • armclang Integrated Assembler.
Clarified the information about image entry points and dealing with multiple entry points.	• -e. • --entries. • ENTRY directive. • --entry=location. • --keep=section_id (armlink). • Image entry points. • The initial entry point for an image.
Improved the description and examples for the `armclang` option `-E` to provide more detail on how to pre-process source files and scatter files. Also added a link to new User Guide topic on how to use this feature to provide simplified source code to Arm support.	• -E.
Clarified that the `-Omin` option does not provide the minimum code size.	• -O (armclang) • --lto_level • -Omin (armlink)
Added description of the new `armclang` options `-mrestrict-it` and `-fno-restrict-it`.	• -mrestrict-it, -mno-restrict-it.
SVE auto-vectorization is supported in 6.18, but without SVE optimized libraries.	• -fvectorize, -fno-vectorize • -O (armclang)
Added warnings that using the [COMMUNITY] option `-fsized-char` can cause problems.	• -fsigned-char, -funsigned-char • Characters • Support level definitions

Table 6: Issue 0619-00

Change	Topics affected
Added a description of the `armclang` options `-feliminate-unused-debug-types` and `-fno-eliminate-unused-debug-types` as a [COMMUNITY] feature.	• -feliminate-unused-debug-types, -fno-eliminate-unused-debug-types.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Added a note that the `armasm` legacy assembler is deprecated.	• Arm Compiler for Embedded tool command-line syntax. • armasm Legacy Assembler Reference. • Using armasm. • *armasm command-line syntax*. • armasm Command-line Options.
Added a description of the `armclang` option `-finstrument-functions` and the `__attribute__((no_instrument_function))` function attribute as [COMMUNITY] features.	• -finstrument-functions. • __attribute__((no_instrument_function)) function attribute.
Added a note that using manual and automatic overlays within the same program is not supported.	• --overlay_veneers. • Execution region attributes.
Added support for *Control Flow Integrity* (CFI) sanitizer schemes.	• armclang Command-line Options. • -fcomplete-member-pointers. • -fsanitize, -fno-sanitize. • -fsanitize-ignorelist, -fno-sanitize-ignorelist. • -resource-dir. • armclang Command-line Options. • __attribute__((no_sanitize("option"))) function attribute
Added support for shadow call stack.	• -ffixed-x18. • -fsanitize, -fno-sanitize. • __attribute__((no_sanitize("option"))) function attribute.
Added support for *Undefined Behavior Sanitizer* (UBSan) checks.	• -fsanitize, -fno-sanitize. • -fsanitize-ignorelist, -fno-sanitize-ignorelist. • -fsanitize-minimal-runtime. • -fsanitize-recover, -fno-sanitize-recover. • -fsanitize-trap, -fno-sanitize-trap. • armclang Command-line Options. • __attribute__((no_sanitize("option"))) function attribute
Added support for Armv8-R AArch64 with hardware floating-point.	• -march. • --cpu=name (armlink). • --cpu=name (fromelf).
Clarified when `armclang` passes options to `armlink`.	• -flto, -fno-lto. • -L. • -l. • -nostdlib. • -u (armclang).
Updates for heap memory tagging.	• --library_security=protection.
Added descriptions of the `armlink` options `--check_pac_mismatch` and `--info=pac`.	• --check_pac_mismatch. • --info=topic[,topic,...] (armlink).
`--mcmodel=large` is now fully supported.	• -mcmodel.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Added note that build attribute compatibility checking is supported only for AArch32 state.	• --cpu=name (armlink).
Added caution about suppressing messages.	• --diag_suppress=tag[,tag,...] (armasm). • #pragma clang diagnostic. • --diag_suppress=tag[,tag,...] (armlink).
Added a description of the `armclang` option `-mframe-chain`.	• -fomit-frame-pointer, -fno-omit-frame-pointer • -mframe-chain. • armclang Command-line Options.
Added architecture feature support for the Cortex-M85 processor.	• Supported architecture feature combinations for specific processors.
Added note about Cortex-M55 processor feature support.	• -mcpu.
Added a description of the `armclang` option `-mtune` as a [COMMUNITY] feature.	• -mtune=target. • armclang Command-line Options.
Mentioned that `-mpure-code` is an alias for `-mexecute-only`.	• -mexecute-only. • armclang Command-line Options.
Added information for the `relro` option to `#pragma clang section`.	• #pragma clang section
Corrected description of 32-bit general purpose registers W0-W30. Previously the text mentioned W0-W31 in error (there is no W31). Also added a note to clarify that W<n> refers to 32-bit registers while X<n> refers to 64-bit registers.	• Constraint codes for AArch64 state.
Added information about dealing with leftover debug data for code and data removed by `armlink`.	• --dangling-debug-address=address.
Fixed the stack protection examples.	• -fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector. • -Rpass.
Updated the topics for `__at` sections.	• __attribute__((section("name"))) function attribute. • __attribute__((section("name"))) variable attribute. • Automatic placement of __at sections. • Placement of __at sections at a specific address.
Added descriptions of the `armlink` options `--elf-output-format`, `--scatterload-enabled`, and `--no-scatterload-enabled`.	• --elf-output-format. • --scatterload-enabled, --no-scatterload-enabled.
Improved and expanded the information for `armclang` option `-W`.	• -W (armclang).
Clarified the information for the `armclang` option `-fno-builtin`.	• -fno-builtin.
Updated the floating-point information.	• -mcpu.
Updated the information for `Region$$Table`.	• --rwpi.
Added [BETA] support for the Armv8.9-A and Armv9.4-A architectures.	• -march. • -mcpu.
Corrected the description of the `__SOFTFP__` predefined macro.	• Predefined macros.
Added a description for the `armclang` options `-mglobal-merge`, `-mno-global-merge`.	• -fdata-sections, -fno-data-sections. • -mglobal-merge, -mno-global-merge. • armclang Command-line Options.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Added information for DWARF 5 support.	• -g, -g<level>, -gdwarf-<version> (armclang). • armclang Command-line Options.
Added information about C++17 support.	• -std.
Added notes about use of the `.w` width specifier.	• armclang inline assembler. • armclang Integrated Assembler.
Added the *Useful resources* topic.	• Useful resources.

Table 7: Issue 0620-00

Change	Topics affected
Improved the description of the `armclang` option `-mframe-chain`.	• -mframe-chain.
Updated the default C++ language standard.	• -std.
Updated the description of the `ALIGN` directive.	• ALIGN directive.
Added a note about infinity and NaN to `armclang` option `-ffp-mode`.	• -ffp-mode.
Updated the description of `t` constraint code.	• Constraint codes for AArch32 state.
Added a note that any undocumented C and C++ language standards are [COMMUNITY] features.	• -std.
Clarified support for `__thread` and `thread_local` keywords.	• -ftls-model. • __attribute__((tls_model("model"))) variable attribute.
Improved the description of weak references and weak definitions.	• Weak references and definitions. • __attribute__((weak)) variable attribute. • __attribute__((weak)) function attribute.
Added documentation for the `armclang` option `-ffp-contract`.	• -ffp-contract.
Updated the description of `#pragma STDC FP_CONTRACT`.	• Floating-point.
Added a description for `#pragma STDC`.	• #pragma STDC.
Removed the `--emit=got` option.	• --emit=option[,option,...].
Added details for `+fp` feature to floating-point extensions table for AArch32 state.	• -mcpu.
Added details about floating-point exceptions support.	• -ffp-mode.
Updated the description for M-profile dynamic linking.	• -fsysv, -fno-sysv. • --sysv.
Added the description of the `+&` constraint modifier.	• Constraint modifiers.
Added the default type for the `.type` directive.	• Type directive.
Added details about the changes to the `-fsanitize=memtag` option.	• -fsanitize, -fno-sanitize. • --library_security=protection.
Added that Armv8.9-A and Armv9.4-A architectures are now fully supported.	• -march. • -mcpu.
Added more details about the available options for the SME and SVE extensions.	• -mcpu.
Updated the description of the AArch64 target selection directives.	• AArch64 target selection directives.
Added support for new ACLE defined Special register intrinsics.	• ACLE support.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Added details of the new `__ARM_FEATURE_SYSREG128` predefined macro.	• Predefined macros.
Added [ALPHA] support for the `b16b16` feature.	• -mcpu.
Added that the `cssc`, `d128`, `ite`, `lse128`, `predres2`, `rasv2`, `rcpc3`, and `the` features are now fully supported.	• -mcpu.

Table 8: Issue 0621-00

Change	Topics affected
Clarified the description of `__attribute__((naked))` function attribute.	• __attribute__((naked)) function attribute.
Added architecture feature names in the list of architectures supported by `-march` and `-mcpu`.	• -march. • -mcpu.
armclang does not generate `.constdata` sections, so removed it from the *Relationship between the default armclang-generated sections and input section selectors* table.	• Relationship between the default armclang-generated sections and scatter-loading input sections.
Added documentation for the `__attribute__((common))` and `__attribute__((nocommon))` variable attibutes.	• -fcommon, -fno-common. • __attribute__((common)), __attribute__((nocommon)) variable attribute. • Compiler-specific Function, Variable, and Type Attributes.
Documented the incompatibility of *eXecute-Only* (XO) functions and the function sanitizer.	• -fsanitize, -fno-sanitize. • -mexecute-only.
Corrected the information on importing linker-defined symbols.	• Methods of importing linker-defined symbols in C and C++.
The *Link-Time Optimization* (LTO) `.llvmbc` section has been renamed to `.llvm.lto`.	• --lto, --no_lto. • Relationship between the default armclang-generated sections and scatter-loading input sections
XO is supported on Armv6-M.	• --execute_only. • -mexecute-only. • Linker-defined symbols. • --xo_base=address. • Default section placement. • Input sections, output sections, regions, and program segments. • Load view and execution view of an image. • Type 1 image structure, one load region and contiguous execution regions. • Type 2 image structure, one load region and non-contiguous execution regions. • Type 3 image structure, multiple load regions and non-contiguous execution regions. • Placement of functions and data at specific addresses. • Type 1 image, one load region and contiguous execution regions. • Type 2 image, one load region and non-contiguous execution regions. • Type 3 image, multiple load regions and non-contiguous execution regions.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Added the relocation specifiers for Armv6-M.	• Assembly expressions.
Updated the *Arm C Language Extensions* (ACLE) support information.	• ACLE support.
Moved the content of *Summary of armclang command-line options*.	• armclang Command-line Options.
Moved the content of *Function attributes*, *Type attributes*, and *Variable attributes*.	• Compiler-specific Function, Variable, and Type Attributes.
Moved the content of *Keyword extensions*.	• Compiler-specific Keywords and Operators.
Moved the content of *Overview of via files*.	• Via File Syntax.
Added information for the `armclang` option `-nobuiltininc`.	• armclang Command-line Options. • -nobuiltininc. • -nostdlib. • -nostdlibinc.
Added notes about *eXecute-Only* (XO) with *Position Independent Code* (PIC) for Armv6-M.	• --execute_only. • -mexecute-only. • Linker-defined symbols. • --xo_base=address. • Default section placement. • Input sections, output sections, regions, and program segments. • Load view and execution view of an image. • Type 1 image structure, one load region and contiguous execution regions. • Type 2 image structure, one load region and non-contiguous execution regions. • Type 3 image structure, multiple load regions and non-contiguous execution regions.
Added documentation for the `-freestanding` option.	• armclang Command-line Options. • -ffreestanding. • -fno-builtin.
Added restrictions for `-fno-rtti`.	• -frtti, -fno-rtti.
Added note about *big-endian* (BE) and `arm_sve.h`.	• ACLE support.
A32 and T32 instruction substitutions has been moved from the *Migration and Compatibility Guide*.	• A32 and T32 instruction substitutions.
Clarified the description of *Unified Assembler Language* (UAL) and A64 in Armv8.	• Syntax differences between UAL and A64 assembly language
Clarified the software *floating-point* (FP) support in AArch64.	• --fpu=name (armasm). • --fpu=name (armlink). • --fpu=name (fromelf).
Added note that Arm Compiler for Embedded 6 does not support flushing denormal half-precision values.	• -ffast-math, -fno-fast-math. • -ffp-mode. • -mcpu. • Half-precision floating-point data types. • Half-precision floating-point number format.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Updated the information for the `-fsanitize` and `-fno-sanitize` options	• -fsanitize, -fno-sanitize. • Examples for the armclang -fsanitize command-line option. • -fsanitize-minimal-runtime. • -fsanitize-recover, -fno-sanitize-recover. • __attribute__((no_sanitize("option"))) function attribute.
Fixed errors in the `armlink --be32` example command line.	• --be32.
Enhancements to the `__attribute__((aligned))` type and variable attribute descriptions.	• __attribute__((aligned)) type attribute. • __attribute__((aligned)) variable attribute.
Updated the description of `fromelf --cpu` for AArch64.	• --cpu=name (fromelf).
Clarified the C and C++ language support and libc++ library support.	• -std.
Improvements to the information on demand paging.	• --paged. • --pagesize=pagesize. • Linker support for creating demand-paged files. • AlignExpr(expr, align) function. • GetPageSize() function. • SizeOfHeaders() function. • Alignment of regions to page boundaries.
Documented support for the *Straight-Line Speculation* (SLS) mitigation option `-mharden-sls`.	• -mharden-sls.
Documented support for the return address signing hardening option `-mharden-pac-ret`.	• -mharden-pac-ret.
Added information for the `harden-pac-ret` option to `__attribute__((target("options")))`.	• __attribute__((target("options"))) function attribute.
Added that the 64-bit Armv9.5-A architecture support level is [BETA].	• -march.
Added [ALPHA] support for the `cpa` and `fp8` features.	• -mcpu.
Added [BETA] support for the `pauth-lr` and `tlbiw` features.	• -mcpu.
Added information for the `pc` option to `-mbranch-protection`.	• -mbranch-protection.
Added a note that the Base Platform linking model is deprecated.	• Base platform linking model overview.
Added a note that the *Base Platform Application Binary Interface* (BPABI) is deprecated.	• Base platform application binary interface BPABI linking model overview.

Table 9: Issue 0622-00

Change	Topics affected
Added a description of the `-faligned-new` compiler option.	• -faligned-new.
Added descriptions of the `-foptimize-sibling-calls` and `-fno-optimize-sibling-calls` compiler options.	• -foptimize-sibling-calls, -fno-optimize-sibling-calls.
Added a description of the `-fstack-usage` compiler option.	• -fstack-usage.
Added a description of the `-isystem` compiler option.	• -isystem <directory>.
Added a description of the `-mabi=<name>` option.	• -mabi=<name>.
Added a descriptions of the `-mdefault-build-attributes` and `-mno-default-build-attributes` compiler options.	• -mdefault-build-attributes, -mno-default-build-attributes.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Added notes about not being able to use the LTO intermediate file as an input to other link commands.	• --keep_intermediate. • --lto_intermediate_filename.
Removed the statement for Armv8.6-A and later, that `+sve` also enables `+f32mm` (FEAT_F32MM).	• Scalable Vector Extension.
Updated the scatter file example for the `--keep=<section_id>` linker option.	• --keep=section_id (armlink).
Improved the example for the `-fsanitize-trap` compiler option.	• -fsanitize-trap, -fno-sanitize-trap.
Added details about how `-ffp-mode` affects the value of `math_errhandling`.	• -ffp-mode. • Library functions.
Added the `arm` and `thumb` options for the `__attribute__((target("<options>")))` function attribute.	• __attribute__((target("options"))) function attribute.
Updated the examples for the alignment related attributes.	• __attribute__((aligned)) type attribute examples. • __attribute__((aligned)) type attribute. • __attribute__((aligned)) variable attribute.
Added a description for the `__attribute__((optnone))` function attribute.	• __attribute__((optnone)) function attribute.
Improved the description of the `--elf-output-format` linker option.	• --elf-output-format.
Added and updated the descriptions for Armv8-R AArch64 software floating-point support.	• -march. • -mcpu. • Floating-point extensions.
Added the `IT` and `SEVL` instructions to describe the `armasm`-specific behavior.	• armasm-Specific A32 and T32 Instruction Set Features. • IT instruction (A32). • SEVL instruction (A32).

Table 10: Issue 0623-01

Change	Topics affected
Added a caution aboout use of invalid feature combinations.	• -march. • -mcpu.
-ftrivial-auto-var-init=zero is now fully supported.	• -ftrivial-auto-var-init.
Updated the default information for the `-munaligned-access` and `-mno-unaligned-access`.	• -munaligned-access, -mno-unaligned-access.
Added notes about using the `-fno-omit-frame-pointer` option with the `armlink` options `--callgraph`, `--info=stack`, or `--info=summarystack`.	• -fomit-frame-pointer, -fno-omit-frame-pointer. • --callgraph, --no_callgraph. • --info=topic[,topic,...] (armlink).
Added information about `basic_stringbuf` default constructor `stringbuf.cons` uses the *Small String Optimization* (SSO) (31.8.2.2).	• Input/output library.
Added a note about removed support for C++98 and C++03 in a future release.	• -std.
Updated the architecture features.	• -march. • -mcpu.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Change	Topics affected
Corrected the translation limits error message.	• Translation limits
Added restriction about interrupt `ABORT`.	• __attribute__((interrupt("type"))) function attribute.
Updated the extension for Armv9.6-A support.	• -march. • -mbranch-protection. • Matrix Multiplication extension. • Scalable Matrix Extension. • Scalable Vector Extension.
Added an example showing the differences between `__fp16` and `_Float16` at `-O1` and higher.	• Half-precision floating-point data types.
Added information about using NaN or infinity for full IEEE.	• -ffp-mode.
Added information about file-scope inline assembly and LTO.	• File-scope inline assembly. • Inline Assembly. • -flto, -fno-lto.
Updated the support levels for Armv9.6-A features.	• -march. • -mcpu.
Updated the support levels for `+sme2p1` (FEAT_SME2p1) and `+sve2p1` (FEAT_SVE2p1).	• -march. • -mcpu.
Added a description for the `armclang` option `--print-enabled-extensions`.	• --print-enabled-extensions.

Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Convention	Use
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.



We recommend the following. If you do not follow these recommendations your system might not work.



Your system requires the following. If you do not follow these requirements your system will not work.



You are at risk of causing permanent damage to your system or your equipment, or harming yourself.



This information is important and needs your attention.



A useful tip that might make it easier, better or faster to perform a task.



A reminder of something important that relates to the information you are reading.

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm Compiler for Embedded User Guide	100748	Non-Confidential
Arm Compiler for Embedded Migration and Compatibility Guide	100068	Non-Confidential
Arm Compiler for Embedded Arm C and C++ Libraries and Floating-Point Support User Guide	100073	Non-Confidential
Arm Compiler for Embedded Errors and Warnings Reference Guide	100074	Non-Confidential
Arm Support	-	-
Manage Arm Compiler Versions	-	Non-Confidential
User-based licensing User Guide	102516	Non-Confidential
Complex Math Functions	-	Non-Confidential
Complex Matrix Multiplication	-	Non-Confidential
Complex FFT Functions	-	Non-Confidential

Arm® architecture and specifications	Document ID	Confidentiality
Arm Architecture Reference Manual for A-profile architecture	DDI 0487	Non-Confidential
ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition	DDI 0406	Non-Confidential
Arm A64 Instruction Set Architecture	DDI 0596	Non-Confidential
C++ ABI for the Arm Architecture	-	Non-Confidential
C++ Application Binary Interface Standard for the Arm 64-bit Architecture	-	Non-Confidential
ELF for the Arm Architecture	-	Non-Confidential
Exception Handling ABI for the Arm Architecture	-	Non-Confidential
Addenda to, and Errata in, the ABI for the Arm Architecture	-	Non-Confidential
Whitepaper - Armv8-M Architecture Technical Overview	-	Non-Confidential

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential

Non-Arm resources	Document ID	Organization
GCC	-	https://gcc.gnu.org/onlinedocs/gcc
Automatic variable initialization	-	https://reviews.llvm.org
C++ implementation status in LLVM Clang	-	https://clang.llvm.org
CFI directives	-	https://sourceware.org
Controlling Errors and Warnings	-	https://clang.llvm.org
Diagnostic flags in Clang	-	https://clang.llvm.org
Language Compatibility in LLVM Clang	-	https://clang.llvm.org
Shadow Call Stack	-	https://clang.llvm.org
Undefined Behavior Sanitizer	-	https://clang.llvm.org

Copyright © 2019–2024 Arm Limited (or its affiliates). All rights reserved.
Non-Confidential